University of Alberta Library
0 1620 2670168 8

Ex libris
universitatis
albertaensis

UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Yigong Wang

TITLE OF THESIS: Image Synthesis Using Front-to-Back Based Radiosity Methods

DEGREE: Doctor of Philosophy

YEAR THIS DEGREE GRANTED: 1992

PERMISSION IS HEREBY GRANTED TO THE UNIVERSITY OF ALBERTA LIBRARY TO REPRODUCE SINGLE COPIES OF THIS THESIS AND TO LEND OR SELL SUCH COPIES FOR PRIVATE SCHOLARLY OR SCIENTIFIC RESEARCH PURPOSES ONLY.

THE AUTHOR RESERVES OTHER PUBLICATION RIGHTS, AND NEITHER THE THESIS NOR EXTENSIVE EXTRACTS FROM IT MAY BE PRINTED OR OTHERWISE REPRODUCED WITHOUT THE AUTHOR'S WRITTEN PERMISSION.

Date: April 23, 1992

UNIVERSITY OF ALBERTA

Image Synthesis Using Front-to-Back Based Radiosity Methods

by

Yigong Wang

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH IN PARTIAL FULFILLMENT OF THE REQUIREMENT FOR THE DEGREE OF DOCTOR OF PHILOSOPHY.

DEPARTMENT OF COMPUTING SCIENCE

EDMONTON, ALBERTA
SPRING, 1992

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

THE UNDERSIGNED CERTIFY THAT THEY HAVE READ, AND RECOMMEND TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH FOR ACCEPTANCE, A THESIS ENTITLED *IMAGE SYNTHESIS USING FRONT-TO-BACK BASED RADIOSITY METHODS* SUBMITTED BY **YIGONG WANG** IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY.

Date: April 23, 1992

To My Grandma and Grandpa

Abstract

This thesis is concerned with a new radiosity method for image synthesis. In general, a radiosity method includes four steps: polygon subdivision, form-factor calculation, radiosity solution, and image display. The most expensive part is the form-factor calculation. This new radiosity method, termed **FBRM**, uses a front-to-back approach rather than the traditional z-buffer method to calculate form-factors.

In order to determine the depth order of patches efficiently, the subdivision step not only subdivides polygons into patches, but also yields a set of octants by using a linear octree technique. An *octant priority* list with respect to the given viewpoint is obtained by traversing the linear octree. With the octant priority list, the front-to-back based hemi-cube method can speed up form-factor calculation significantly since some of patches that are hidden by other patches can normally be detected and discarded at an earlier stage before they are actually processed.

This thesis also deals with the aliasing problem occurring in calculating form-factors, and the parallel implementation of the **FBRM**. These two aspects are of importance to image quality and efficiency. Another important part of this thesis is the implementation of the **FBRM**. Most of features in the **FBRM** were realized mainly on a Sun 3/60 workstation.

Acknowledgements

This thesis would not be done without many people's help and suggestions. First of all, I like to thank all those unnamed, but important people for their persistent support, encouragement and assistance. I want to thank my supervisor Wayne Davis for his help and guidance throughout my Ph.D program. In particular, I am grateful for his time and patience in reviewing my thesis after his retirement. I also like to thank my supervisory committee members, Barry Joe, Mark Green, Alain Fournier and Ray Rink for their valuable suggestions and comments on how to improve quality of my thesis.

I would like to thank my parents and grandparents for their love, caring and support in all aspects. Very special thanks go to Zheng Youmei for great help and encouragement. I also want to thank Cowper-Smith family who provide such a great environment where I often go to spend my pastime.

Table of Contents

Chapter 1

Introduction 1

1.1 Problems 1

1.2 Contributions 3

1.3 Outline of the Thesis 6

Chapter 2

Review of Related Research 8

2.1 Basic Approaches to Image Synthesis 8

2.1.1 Modelling 8

2.1.2 Hidden Surface Removal 10

2.1.3 Simple Illumination Models 10

2.1.4 Global Illumination - Ray-tracing 12

2.2 Radiosity Methods 15

2.2.1 Thermal Radiation Transfer Theory 15

2.2.2 Previous Work on Radiosity 22

2.2.3 Implementation 26

Chapter 3

3-D Subdivision of Polygons 29

3.1 Introduction 29

3.2 Preprocessing - Shadow Boundary Computation 32

3.2.1 Notations for Polygons 33

3.2.2 Determination of Shadow Boundaries 34

3.3 Octree for 3-D Subdivision 40

3.3.1 Numbering Scheme for Octree 40

3.3.2 Clipping of Polygons 44

3.3.3 3-D Subdivision Algorithm 46

3.4 Specification of Criteria 49

3.4.1 Simple Criteria 49

3.4.2 Criterion Vectors 51

Chapter 4

Visibility in Form-factor Computation 58

4.1 Octant Priority 58

4.1.1 Cluster Priority in Schumacker Algorithm 59

4.1.2 Formation of Octant Priority 60
4.1.2.1 Basic Octant Priority 61
4.1.2.2 Bottom-to-Top Traversal of the Linear Octree 63
4.2 Basis of Hemi-Cube Method 67
4.2.1 Hemisphere 68
4.2.2 Hemi-Cube 70
4.3 Front-to-back Algorithm for Form-Factors 73
4.3.1 Delta Form-factors and 1-bit Mask Table 73
4.3.2 Painting Patches onto the Hemi-cube 75
4.3.3 Why Front-to-Back? 78
4.3.4 Overview of the Algorithm 81
Chapter 5
Anti-aliasing for Form-factors 83
5.1 Problems 83
5.2 Area-weighted Approaches to Anti-Aliasing 86
5.3 Weighted Delta Form-factors 89
5.3.1 Decomposition of Segments 89
5.3.2 Fraction Mask Table 94
Chapter 6
Parallel Implementation of Radiosity 99
6.1 Introduction 99
6.2 Parallel Computation for Form-factors 101
6.3 Implementation on Myrias SPS-2 103
6.4 Parallel Algorithm for Solving Radiosity Equations 105
6.5 More Investigation on Parallelism in Radiosity 107
Chapter 7
Implementation and Results 114
7.1 Implementation of **FBRM** 114
7.1.1 Configuration of Hardware and Software 115
7.1.2 Four Modules of Myrad 119
7.2 Comparison and Results 124
7.2.1 Front-to-back vs z-buffer 124
7.2.2 Crowded Scenes vs Less Crowded Scenes 127
7.2.3 Anti-aliasing vs Aliasing 130
7.2.4 Sequential vs Parallel 133
Chapter 8

Conclusions and Future Work 140
8.1 Summary 140
8.2 Future Work 144
8.2.1 Applying FBRM to Progressive Refinement 144
8.2.2 Effective Region 145
Bibliography 148

List of Tables

Table 1. Basic Octant Priority 63
Table 2. Seven Basic Classes 88
Table 3. Statistical Results for Scene 1 134
Table 4. Statistical Results for Scene 2 134
Table 5. Statistical Results for Scene 3 135
Table 6. Statistical Results for Scene 4 135
Table 7. Statistical Results for Scene 5 135
Table 8. Performance on Distributed Computer 136

List of Figures

Figure 2.1. Radiant energy 16
Figure 2.2. Form-factor 18
Figure 3.1. Regular mesh for polygons 30
Figure 3.2. Polygon subdivision using a quadtree 31
Figure 3.3. Relationship among polygon, lit polygon and patch 34
Figure 3.4. An approximation to the shadow's umbra and penumbra 39
Figure 3.5. Cube subdivision 41
Figure 3.6. The numbering scheme for the octree 42
Figure 3.7. An octree 43
Figure 3.8. Failure of subdivision 50
Figure 3.9. Extensive subdivision 54
Figure 3.10. Two cases of incorrect subdivisions 55
Figure 3.11. An example of polygon subdivision 57
Figure 4.1. Schumacker's cluster priority 60
Figure 4.2. Space subdivision 62
Figure 4.3. The process of octant sort 68
Figure 4.4. Hemisphere approximation 70
Figure 4.5. Hemi-cube approximation 71
Figure 4.6. An example of painting process 75
Figure 4.7. Scan-conversion 77
Figure 4.8. An example of zero form-factors 78
Figure 5.1. Comparison between images with aliasing and without aliasing 85
Figure 5.2. Average intensity 87
Figure 5.3. An example of a segment 89
Figure 5.4. Examples of the segment decomposition 90
Figure 5.5. Failure of segment decomposition 91
Figure 5.6. Cell areas 91
Figure 5.7. Calculating cells in C3 93
Figure 5.8. Calculating cells in C1 and C2 93
Figure 5.9. Calculating cells in C4 and C5 94
Figure 5.10. An example of incorrect visibility computation 98
Figure 6.1. Partitioning of pixels 100
Figure 6.2. Three loops in the radiosity equations 106

Figure 6.3. Pixel partitioning on the hemi-cube 108
Figure 6.4. The configuration of the parallel system..... 110
Figure 6.5. The view volume piercing through the small region 111
Figure 7.1. The overall organization..... 115
Figure 7.2. An example of object hierarchy..... 118
Figure 7.3. The data flow between the four modules..... 120
Figure 7.4. Empirical analysis of time complexity 121
Figure 7.5. Several cases of configuration..... 122
Figure 7.6. Comparison between z-buffer and front-to-back..... 127
Figure 7.7. Comparison between crowded and less crowded scenes..... 130
Figure 7.8. Miscalculation of delta form-factors because of aliasing 132
Figure 7.9. Comparison between sequential and parallel..... 133

List of Color Plates

Color Plate 1 The image with anti-aliasing for Scene 1 137
Color Plate 2 The image with anti-aliasing for Scene 2 137
Color Plate 3 The image with anti-aliasing for Scene 3 138
Color Plate 4 The image with anti-aliasing for Scene 4 138
Color Plate 5 The image without anti-aliasing for Scene 5 139

Chapter 1

Introduction

Realistic image synthesis plays an important role in today's computer graphics. Advanced development of this technology will undoubtedly reduce the production cost of graphics applications, especially real-time computer-based animation and simulation. Unfortunately, current technology in image synthesis is still far from this goal. An underlying problem is how to balance the trade-off between image quality and display efficiency. The radiosity method was developed to enhance image quality. Although the early radiosity methods were inefficient, research in this area has been conducted for years to speed up radiosity image generation. In this chapter, the central problems in the radiosity area are identified, and an outline of the thesis is given.

1.1 Problems

Image synthesis is an interesting and broad area involving many disciplines in computer graphics. With the advent of advanced image-based hardware, it has received more and more attention, resulting in many techniques for fast realistic image generation proposed or established in the last two decades. These techniques basically attempt to solve five problems: illumination models, geometric modelling, hidden surface removal, aliasing and texturing.

In image synthesis, illumination is the central problem. An efficient illumination model is demanded in academic research, scientific visualization and the entertainment industry. A number of illumination algorithms such as Phong shading, Gouraud shading,

and the Whitted illumination model, have been used efficiently in graphics designs and realistic image generation. Although efficient, these lighting models simulate part of the light behavior based on simple geometric optics. Ray tracing is a more popular and advanced technique for displaying realistic images, including effects of specular reflection and refraction. However, it is unable to accurately show interreflection of illumination in a 3-D environment. Researchers have tried to overcome this problem and come up with an even more powerful technique known as radiosity. The radiosity method, based on thermal radiation transfer principles in thermal engineering, faithfully simulates characteristics and behavior of lights and generates more realistic and natural images.

The first radiosity approach for image synthesis was proposed by Goral et al. in 1984 [GoTo84]. Since then, many articles to improve this method have been written (for a survey, see Chapter 2). Although the radiosity methods proposed in the past seven years produced many remarkable images, there are many problems still unsolved. This thesis tackles the following four problems.

The first problem is the expensive form-factor calculation. Suppose there are N patches in an environment under consideration. A $\Omega(N^2)$ complexity algorithm is required to calculate $N \times N$ *form factors* between all pairs of patches. A *form factor* represents the geometric relation between two patches. For a formal definition, see Chapter 2. Calculating form factors needs a visibility computation that uses a hidden surface removal algorithm. So far, the widely adopted algorithm is the z-buffer because of its simplicity. It is natural to consider other sophisticated methods for hidden surface removal involved in radiosity. Unfortunately, little work has been done to use better alternatives for fast hidden surface removal.

Secondly, while the speed for form factor computation is the main concern, image quality is another problem. The z-buffer has difficulty dealing with aliasing problems. Various aliasing phenomena occur in computing form factors. They cause serious artifacts

on images. This implies that from either the quality or efficiency aspect, more sophisticated hidden surface removal algorithms rather than the simple z-buffer have to be used in the radiosity method.

The third problem is the assumption, required by any radiosity method, that each patch involved has constant intensity. This assumption makes it difficult to decide how many patches a surface should be subdivided into so that each patch has a constant intensity. It is always true that the more patches there are, the better image quality will be produced. But too many patches require more computation. Polygon subdivision should be, therefore, performed in the way adaptive to the intensity gradients. In other words, polygons whose intensities change dramatically across their surfaces should be partitioned into finer patches than those whose intensities vary gradually. Adaptive subdivision of polygons is desired, but difficult to implement without a complicated mechanism. In particular, achieving a perfectly adaptive subdivision is almost impossible. It is, however, possible to obtain an adaptive subdivision that is not expensive but yields good enough results. This is one of the goals that this thesis is attempting to achieve.

A fourth aspect is the parallel implementation of radiosity. Radiosity, especially its form-factor determination, is computationally intensive. Presently, parallel radiosity algorithms have rarely been seen in the literature. It is worthwhile to investigate parallelism of radiosity and study the possibility of implementing radiosity on parallel processing systems.

1.2 Contributions

This thesis basically attempts to solve the four problems described above, by focusing on the first three. Based on previous work, especially the hemi-cube method [CoGr95], a new radiosity approach is developed. Several images are produced by implementing this new method. This thesis also includes an in-depth analysis of radiosity in many aspects,

and explores the importance and potential of the new radiosity method. The main contributions of this thesis can be summarized as follows. For descriptive convenience, this new method is termed **FBRM** (**F**ront-to-**B**ack based **R**adiosity **M**ethod) throughout this thesis to distinguish it from previous methods.

1. Octree techniques are used for the first time in radiosity methods. They are integrated with radiosity methods to perform polygon subdivision. An octree is usually used to partition a particular space into ordered sub-spaces to represent objects. The octree method in **FBRM** not only performs the space partitioning, but it also serves an additional function: polygon subdivision. The dividing planes for the polygon subdivision are those virtual planes separating sub-spaces in the octree. The octree based subdivision is called 3-D subdivision.

2. 3-D subdivision can not only subdivide polygons into patches, but also perform adaptive subdivision around shadow boundaries and object corners. A shadow algorithm is used to output a list of lit polygons at a preprocessing step. The preprocessing only provides shadow information from the subpolygons; then the 3-D subdivision uses this information to generate an adaptive subdivision around the shadows.

3. With the introduction of octrees, a new concept, *octant priority*, is established, which plays a crucial part in the **FBRM**. In order to determine the visibility of patches efficiently, sub-spaces (called *octants* in **FBRM**) are sorted for any given viewpoint. This is done by traversing the octree. Once the octant priority is formed, a hidden-surface removal algorithm is used to scan patches from the nearest to the farthest.

4. The front-to-back paint algorithm replaces the z-buffer for visibility calculation in determining form factors. With the front-to-back algorithm, the hemi-cube

method is modified in order to save memory and time. Neither the z-buffer nor the item-buffer, required in a conventional hemi-cube, are needed.

5. A 1-bit mask table that plays a role similar to the item-buffer is introduced to assist in determining visible parts of patches and accumulate delta form factors. A traditional hemi-cube method obtains form factors by first scanline-converting patches into the frame buffer and then accumulating the delta form factors for each patch after all patches in the scene are converted. The **FBRM** with the 1-bit mask table, however, accumulates delta form factors for a patch as the conversion for the patch is proceeding. A big advantage of this scheme is that when the mask table is full[1], there is no need to continue to process the remaining patches because they must be occluded by the patches that have been processed.

6. In addition to the saving of memory and time, the 1-bit mask table has another advantage. It allows the radiosity method to handle aliasing problems occurring on the hemi-cube during form factor calculation by extending it to a fraction mask table where each element of the table holds a floating point number. In the traditional hemi-cube method, aliasing was hard to remove because of the nature of the z-buffer. The **FBRM** scheme solves the problem nicely, but with some speed loss.

7. The **FBRM** not only proves that other approaches are possible to handle form factors, but also forms a basis for more advanced radiosity methods. This thesis describes other advanced features that could be added to **FBRM** to improve the performance of radiosity methods.

   • Parallel implementation methodology for radiosity methods was studied in depth. This study reveals that there is a significant amount of parallelism in

---

[1]The meaning of fullness will be defined later

radiosity methods. Several levels of parallelism were explored and the corresponding parallel algorithms are given in this thesis. Two of the parallel algorithms were implemented on the Myrias SPS-2 supercomputer with 64 PEs. Although primitive, the work is valuable to future improvement and pioneering in the area of parallel radiosity methods.

- The **FBRM** is an alternative approach to radiosity images that is flexible enough to be a substitute for the hemi-cube. The **FBRM** can be applied to the progressive refinement method that uses the hemi-cube as the core to calculate form-factors. The combination of the **FBRM** and the progressive refinement, which takes the advantages of both methods, yields even better performance for radiosity image generation. This thesis describes how to apply the **FBRM** to the progressive refinement method and points out the significant improvements achieved by the combination of the two methods.

- *Effective region* is introduced in this thesis. The use of this important concept will save a great amount of computation for the form factors that can be predicted to be zero. It is observed that a less-reflective or less-emitting object contributes light only within a local area. In other words, any other object B far from the object A under consideration receives little light emitted from A. This implies that the form factor from A to B can be assumed to be zero and the calculation for this form factor can be eliminated. **FBRM** can handle this case easily by discarding all the patches whose distances from the viewer are greater than a given threshold.

1.3 Outline of the Thesis

Chapter 2 conducts a brief review of background material necessary to understand the rest of the thesis. Three related aspects that will be discussed are traditional image synthesis

techniques, thermal radiation transfer, and the previous radiosity methods. In particular, the theoretical background for radiosity is described in detail.

In Chapter 3, a 3-D octree based subdivision of polygons is introduced. The preprocessing for clipping shadowed polygons into lit polygons is described first. An algorithm for octree construction and polygon subdivision is then developed. Various subdivision termination criteria are discussed.

Based on octant priority, Chapter 4 introduces a modified hemi-cube method using a front-to-back paint algorithm for form factor calculation. The difference between the front-to-back and the z-buffer methods are explained and the importance of the 1-bit mask table, a technique assisting determination of visibility, is emphasized in this chapter.

Chapter 5 develops the mask table technique to accommodate the anti-aliasing method for obtaining more accurate form factors. The concept of a fraction mask table is introduced. A detailed algorithm for calculating anti-aliasing form factors is shown at the end of the chapter.

In Chapter 6, parallelism in the radiosity method is studied in depth. Parallel algorithms for different components in **FBRM** are developed and proposed, two of which are described in detail by showing their implementation results.

Chapter 7 is concerned with implementation of the **FBRM**. Results produced by implementation are collected and compared. Several images with statistical data are shown. Advantages and disadvantages are discussed by comparing the new method with traditional methods.

Chapter 8 concludes the thesis by summarizing the main aspects appearing in the thesis and providing suggestions for further improvement of radiosity methods.

Chapter 2

Review of Related Research

Image synthesis involves many areas not only in computer graphics, but also in physics. These areas include geometric modelling, hidden surface removal, and illumination models. Knowing the background materials related to these subjects is necessary to understand the mutual dependency and relationship between them. An ideal illumination model, that combines simple and efficient geometric object representation and fast visibility computation, would significantly improve image generation. Physics provides a theoretical basis on which an illumination model is developed. Thermal radiation transfer theory plays an important part in developing the radiosity method. Therefore, this chapter first reviews several typical techniques for image rendering, and then introduces the principles of thermal radiation transfer.

2.1 Basic Approaches to Image Synthesis

An image synthesis method is mainly concerned with the following problems: how to represent geometric objects, how to determine visibility of objects, and how to display objects. Many algorithms have been developed for solving these problems. This section highlights major algorithms involved in modelling, hidden surface removal and illumination.

2.1.1 Modelling

Geometric modelling is concerned with the computer representation of geometric objects. Currently, there are many sophisticated geometric modeling techniques. The most common

methods include parametric representation, polygon mesh techniques, procedural methods and space subdivision schemes. Since the **FBRM** uses a 3-D subdivision algorithm, the discussion is focused on the octree technique, a space subdivision method.

An octree is a natural extension of a quadtree that is used to represent images on the 2-dimensional plane. One of the earliest publications concerned with quadtree techniques goes back to 1972 [SiBo72]. Since then, many papers have been published. A survey of quadtree techniques can be found in [Ro80]. Based on quadtrees, octrees were introduced and developed by Jackins and Tanimoto [JaTa80].

A conventional model represents a three-dimensional object by describing the vertices of the polygons forming the object in the form of (x,y,z). In contrast to the traditional method, an octree model represents a three-dimensional object by describing the space occupied by the object in a form of tree with each node corresponding to a number 0 or 1. A node is assigned a 1 if the space associated with the node is "solid matter"; a node is assigned a 0 if it is empty. A fundamental octree technique is subdivision: dividing the whole scene of objects into eight cube-shaped arrays of equal size, and recursively continuing this subdivision on each of the eight cubes, respectively, until each cube contains either "nothing" or "solid matter". Any region obtained in each step of this subdivision process is called an octant. The final representation of a scene is a tree structure in which each node is associated with an octant.

One of the advantages of the octree-based modelling is that the geometric primitives appear in the form of regular-shaped voxels. This kind of regular-cell representation is easy to render, and is suited for implementation by parallel algorithms or a parallel architecture as well. More importantly, these voxels can be sorted efficiently in the space by traversing the octree. Ordering of primitives is very important to speed up rendering, and especially hidden surface removal. As a result, these advantages have made octrees popular in recent years. An octree model, however, is only an approximate

representation of objects. It is difficult and expensive to model a complex scene using an octree.

2.1.2 Hidden Surface Removal

Hidden-surface determination is always a difficult procedure in rendering three-dimensional objects. Many earlier algorithms for hidden-surface and hidden-line removal, from simple z-buffer methods to complex area-subdivision algorithms, are contained in [FoDa83]. Of these algorithms, the z-buffer approach is one of the most practical algorithms, due mainly to its simplicity. In [Ro85], many other algorithms are described in detail. These algorithms include the Roberts algorithm, the Warnock algorithm, the Weiler-Atherton algorithm, the scan line z-buffer algorithm, the spanning scanline algorithm and the ray tracing method. A survey on hidden surface removal appeared in [SuSp74] that characterizes ten hidden surface removal algorithms by comparing them from many different points of view.

Although there are basic differences between various hidden-surface algorithms, most of these algorithms use sorting and coherence methods to improve performance. Sorting is the key to performing depth comparisons for determining visible surfaces. Coherence methods take advantage of regularities of a scene to speed up the rendering process. For example, an individual scanline can contain intervals of constant pixel intensities, and scan-line patterns often change little from one line to the next. Thus the next pixel or line can be inherited from the previous one without re-calculation from scratch. As a result, seeking fast sorting techniques and effective coherence methods has been the main focus for years in hidden surface removal.

2.1.3 Simple Illumination Models

Hidden-surface removal is not enough to display a realistic 3-D image. Shading and color patterns need to be added to the visible surfaces in order to give more realism. To apply a

shading model, light sources and the viewpoint must be specified. Shading techniques simulate the effect of light sources on objects by using basic principles of optics.

Although from the optics point of view, shading is a very complicated process, the shading formula in computer graphics are normally simplified to include only four elements: ambient illumination, diffuse reflection, specular reflection and refraction, each quantified by intensity. Different lighting models interpret the effects of light to a viewer in various ways, depending on how the four elements are formulated. A commonly used simple model can be defined by:

$$I = I_a K_a + \frac{I_p}{d + d_0}[K_d(\overline{L}\cdot\overline{N}) + W(\theta)(\overline{R}\cdot\overline{V})^n]$$

where the first term is ambient light, the second diffuse reflection and the last specular reflection[1]. *Ia* and *Ip* stand for intensity values of ambient light and the point light source, respectively. *Ka*, *Kd* and $W(\theta)$ characterize reflectivities of ambient, diffuse, and specular light from the object's surfaces, where *Ka* and *Kd* are constants and θ is the angle between the incident light and the surface normal. Very often, $W(\theta)$ is also set to a constant *Ks*. d_0 is a constant and d is the distance from the viewpoint to the surface. The value of n typically varies from 1 to 200, depending on the surface. The four vectors $\overline{L}$, $\overline{N}$, $\overline{R}$ and $\overline{V}$ represent the direction to the point light source, the normal to the surface, the direction of specular reflection and the direction to the viewpoint, respectively. Calculating these vectors is the main task for determining the intensity *I*.

The three basic ways to shade objects represented by polygon meshes are constant shading, intensity interpolation shading, and normal-vector interpolation shading. The first shading is straightforward, without considering intensity changes across surfaces. For the second case, the normal calculation is simple, but the intensities of a polygon are not obtained by the derived normals, but by interpolating intensities in a way that the

---

[1]The model does not consider refracted light.

intensity values cross the plane boundaries smoothly. This is called Gouraud shading [Go71]. Phong shading obtains surface normals by interpolating normal vectors first along edges and then along a scan line, and finally uses the surface normal at each pixel to determine the intensity of the pixel [Po73].

Gouraud shading removes intensity discontinuities associated with the constant shading model, but it has some deficiencies. Highlights on the surface are sometimes displayed with anomalous shapes, and linear intensity interpolation can cause bright or dark intensity streaks, called Mach bands, to appear on the surface. The Phong shading method solves most of these problems, but is still a linear interpolation scheme. Consequently, discontinuities in the first derivative of the intensity still give rise to Mach band effects. In addition, Phong shading requires more complicated computation than Gouraud shading.

Simple illumination models only consider the contribution of a light source. Global illumination models discussed in the next subsection produce more realistic images of three-dimensional objects by interpreting illumination emitted directly from the light source and reflected from the environment. There are basically two approaches that are widely used for displaying realistic pictures with global visual effects. Ray-tracing is a well-known technique widely used in computer animation; radiosity is a new method presently being used only in experiments.

2.1.4 Global Illumination - Ray-tracing

Ray-tracing techniques were first introduced by Turner Whitted [Wh80]. Basically, the intensity of the reflected light at each point (pixel) in an image is determined by tracing light rays from the viewer through the scene in ray-tracing.

There are two processes going on in a ray trace. The first constructs a ray tree as follows: for each pixel in an image, a ray is launched in the direction from the viewpoint to the pixel, the intersection of the ray with the first polygon (the closest to the viewer) is

found, the reflected ray and/or the refracted ray is calculated at the intersection point, and recursively these two rays are traced, respectively, until either all rays have left the scene or the recursive process reaches the given depth. Finally a ray tree is formed in which each node represents an intersection and each branch stands for a ray that links the two adjacent intersections in the ray-tracing process.

The second procedure, starting from the leaves of the ray tree, calculates the intensity at each node according to the intensity returned by its child nodes and the light sources, and returns the intensity value to its parent node. The process continues until it reaches the root of the tree. The intensity at the root node is displayed as the intensity at the pixel under consideration. If the image is a rectangular array of pixels, the entire image is obtained by scanning the array pixel by pixel and performing the ray-tracing operation mentioned above at each pixel.

The biggest problem with ray-tracing is the intersection calculation that takes almost 95% of the total time for a complex scene. As indicated by Whitted, the use of the extent associated with each object significantly reduces the overhead of the intersection calculation. The intersection calculation requires searching of the entire scene in order to find the first polygon that the ray intersects. If each object is bounded within a cube, for example, the intersection test can be performed first against each bounding box. If no intersection is found, the objects within this box are simply discarded because it is impossible for the ray to intersect them. Therefore, the extent method lessens the intersection calculation.

Incremental approaches are also used in order to speed up ray-tracing. A hierarchy based on the octree data structure is suitable for implementing ray tracing algorithms by incremental techniques. Octree data structures for ray tracing were first advocated by Fujimura et al [FuTo83]. Based on the data structure, Fujimura et al. invented a scheme called *3DDDA* (Three-Dimensional Digital Differential Analyzer) to

generate rays on the basis of incremental movement. *3DDDA* adopts a principle similar to *DDA*, the incremental method for straight line generation.

Since the above approaches are based on point-sampling techniques, the images generated are not satisfactory because of aliasing. Cone tracing and distributed ray-tracing are better solutions in terms of quality and realism of images.

Cone tracing was introduced by Amanatides [Am84]. The basic idea behind this approach is that a pixel of an image is defined as an area of the screen instead of a point, and the definition of a ray is changed to a pyramid of rays with the apex at the view point and the base defined by the four planes that cut the borders of the pixel. This cone tracing method has its own advantages. It partially makes use of the coherence property to improve both the image quality and the calculation speed. In addition, this approach produces anti-aliased images, as well as fuzzy images, effectively due to the treatment of a pixel as an area. Since rays are traced through a finite area, the information returned from the reflected and refracted directions is also more accurate.

As far as fuzzy images are concerned, another ray-tracing technique called distributed ray-tracing [CoPo84] rivals beam tracing. By distributing the directions of the rays instead of limiting rays in precise directions, this approach offers correct and easy solutions to some previously unsolved or partially solved problems, such as motion blur, depth of field, penumbras, translucency, and fuzzy reflections. The key to this is the ability to antialias point sampling.

Ray tracing is a very powerful but simple approach to image synthesis. It has many advantages over a scan line method. It allows for both reflection and refraction of light in the same way. In particular, shadow tests can be performed easily by the basic ray-tracing method. The disadvantages are, as indicated above, the expensive intersection calculation and many intensive floating point computations involved.

Although ray-tracing is a global illumination model, it is limited to modelling intra-environment reflections in the specular direction. In addition, since many existing ray tracing systems regard lights as points in space, shadows always exhibit sharp boundaries. Beam tracing and distributed ray tracing improve image quality; however, they still do not sufficiently support more realistic images with complex interreflection of illumination. Furthermore, ray-tracing methods are view-dependent, that is, the entire process must be repeated for every different viewing position. The radiosity method, another global illumination model, however, overcomes these difficulties and opens up a new way for realistic image synthesis.

2.2 Radiosity Methods

The first radiosity method was introduced by Goral et al. in 1984 [GoTo84]. This new methodology for image synthesis is derived from thermal engineering that accurately defines the behavior and formation of visible lights. In order to understand radiosity, the following section explores the principles of thermal radiation transfer. In addition, it gives the preliminary definitions for the derivation of radiosity.

2.2.1 Thermal Radiation Transfer Theory

Light travel can be treated as thermal energy transfer. Energy takes the form of radiant heat. The methodology for calculating radiant heat exchange in various enclosures can be found in [SiHo78, SpCe78]. The main principles related to radiation heat transfer are briefly stated as follows. In an enclosure (an environment enclosed by surfaces), a surface receives energy from the environment and sends energy back to the environment in radiant form. The radiant energy consists of two components: emission and reflection. The reflected energy is proportional to the total energy that the surface receives.

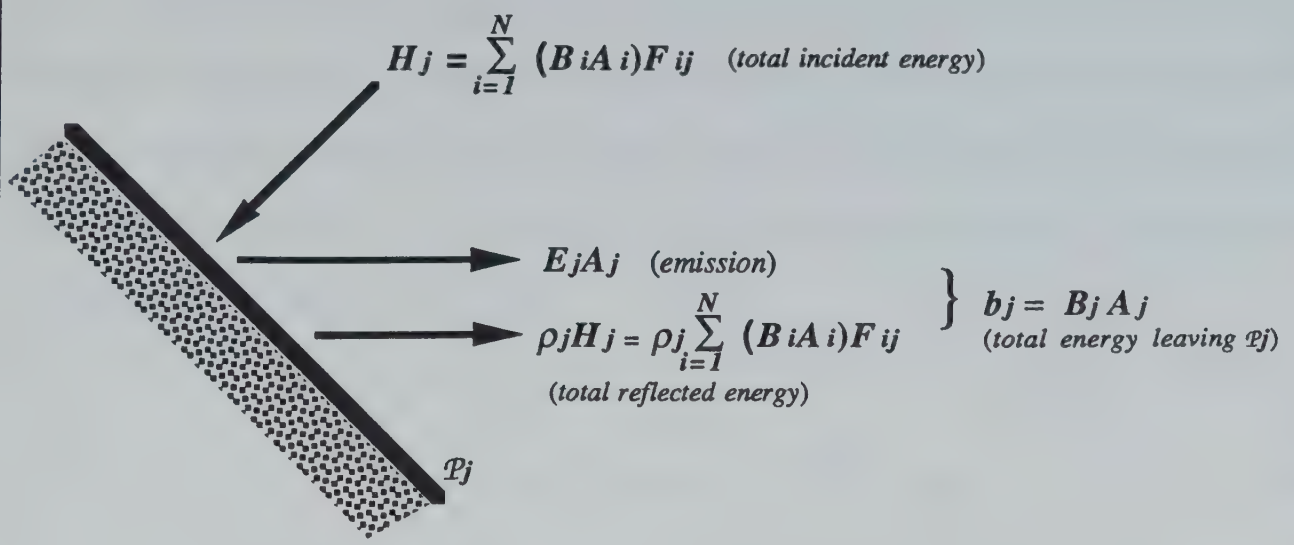

$H_j = \sum_{i=1}^{N} (B_i A_i) F_{ij}$ (total incident energy)
$E_j A_j$ (emission)
$\rho_j H_j = \rho_j \sum_{i=1}^{N} (B_i A_i) F_{ij}$
(total reflected energy)
$b_j = B_j A_j$
(total energy leaving $\mathcal{P}_j$)
$\mathcal{P}_j$

Figure 2.1. Radiant energy. The sum of emission and reflected energy is called radiosity.

If H represents the total energy (watts×unit time) incident to a polygon surface $\mathcal{P}$, and ρ stands for reflectivity – the ratio of the reflected energy to the total incident energy where $0 \leq \rho < 1$, the reflection component of the outgoing energy is, therefore, equal to ρH. Let e represent the energy emitted directly from the surface $\mathcal{P}$ and b be the total energy leaving the surface. Thus the total outgoing energy can be described by $b = e + \rho H$.

For image synthesis, radiant intensity rather than radiant energy is used to display images. Note that the intensity unit is watts per unit time per unit area. Let E stand for the intensity of e, B the intensity of b, and A be the area of $\mathcal{P}$. B is termed the radiosity of $\mathcal{P}$. Thus the total outgoing energy can be rewritten by $B \times A = E \times A + \rho H$ (see Figure 2.1). Note that all these quantities in this equation should be direction-dependent. In other words, the radiant energies vary in different directions. For simplicity, a major assumption must, however, be made that all surfaces are ideal diffuse (i.e., Lambertian) reflectors so that the intensity of outgoing light can be computed independently of the direction of incoming light. This assumption is made throughout this thesis.

Considering two polygons $\mathcal{P}_i$ and $\mathcal{P}_j$, let F_{ij} be the form factor representing the fraction of radiant energy leaving $\mathcal{P}_i$ and impinging on $\mathcal{P}_j$. With these conventions, the total incident energy to the surface $\mathcal{P}_j$ is equal to $H_j = \sum_{1<=i<=N} (B_i \times A_i) F_{ij}$. Note that B_i represents the intensity of $\mathcal{P}_i$, while $B_i \times A_i$ is the total energy leaving $\mathcal{P}_i$ Therefore, the radiant energy leaving $\mathcal{P}_j$ is related by:

$$\begin{aligned} B_j \times A_j &= E_j \times A_j + \rho H_j \\ &= E_j A_j + \rho_j \sum_{i=1}^{N} (B_i A_i) F_{ij} \qquad (j = 1,...,N). \end{aligned} \tag{2.1a}$$

Dividing (2.1a) by A_j, the radiosity B_j is given by:

$$B_j = E_j + \rho_j \sum_{i=1}^{N} B_i \frac{A_i}{A_j} F_{ij} \qquad (j = 1,...,N). \tag{2.1b}$$

As we will see in the following discussion, a reciprocity relationship, $A_i F_{ij} = A_j F_{ji}$, holds between form-factors. Thus,

$$B_j = E_j + \rho_j \sum_{i=1}^{N} B_i F_{ji} \qquad (j = 1,...,N). \tag{2.1c}$$

Since j varies from 1 to N, (2.1c) can be rewritten in the following matrix form.

$$\begin{pmatrix} 1-\rho_1 F_{11} & -\rho_1 F_{12} \ . \ . \ . & -\rho_1 F_{1N} \\ -\rho_2 F_{21} & 1-\rho_2 F_{22} \ . \ . \ . & -\rho_2 F_{2N} \\ \vdots & & \vdots \\ -\rho_N F_{N1} & -\rho_N F_{N2} \ . \ . \ . & 1-\rho_N F_{NN} \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \\ \vdots \\ B_N \end{pmatrix} = \begin{pmatrix} E_1 \\ E_2 \\ \vdots \\ E_N \end{pmatrix} \tag{2.2}$$

(2.2) expresses the dynamic exchange of radiant energies between surfaces in an enclosure. The radiosity of each surface can be obtained by solving the N simultaneous equations with B_j as variables ($j=1,..,N$).

Generally speaking, the reflectivity, ρ_j, and the self-emitted light, E_j, are constants provided with the modelling database. Thus the remaining terms in (2.2) to be determined are the form-factors, F_{ij}, and the unknown variables B_j. Solving the simultaneous

equations for B_j is not a difficult problem. Many methods for solving this equation have been developed in numerical analysis. Particularly, it is found that the Gauss-Seidel iterative method is well-suited for (2.2) because the coefficient matrix is diagonally dominant[1]. Calculating form factors, however, requires a more sophisticated and complicated method.

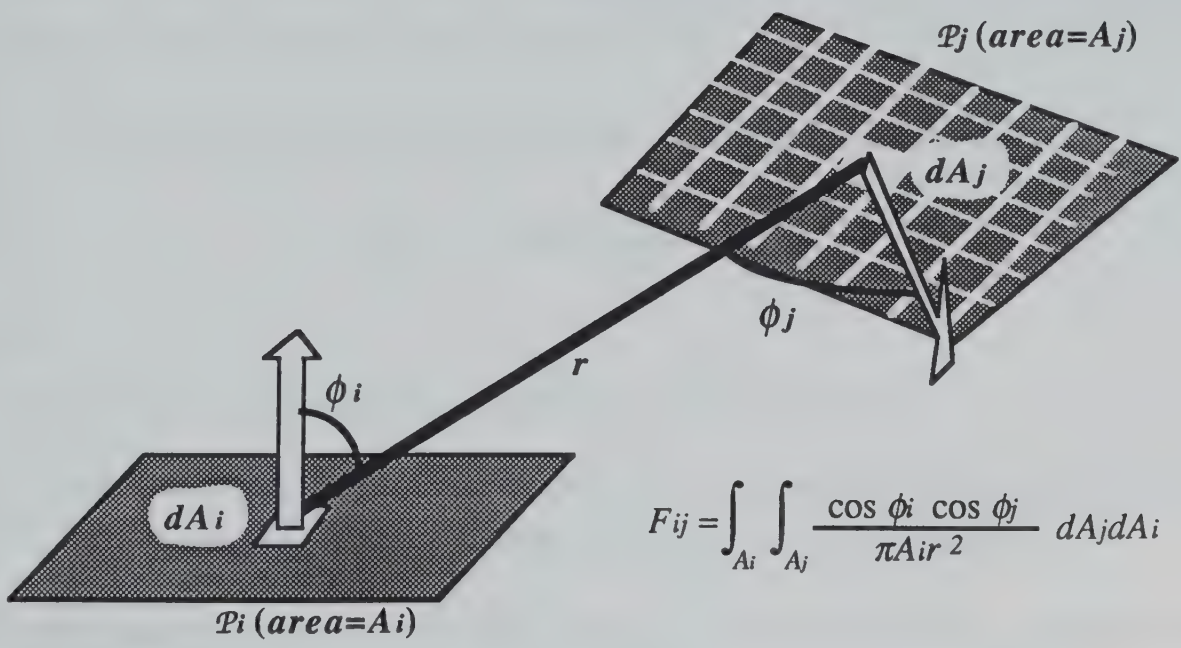


Figure 2.2. Form-factor. Form-factor F_{ij} is a pure geometric relation between two patches.

As mentioned above, the form factor, F_{ij}, represents a fraction of the radiant energy emitted from surface $\mathcal{P}_i$ that lands on surface $\mathcal{P}_j$. From thermal engineering, it is proved that F_{ij} represents a pure geometric relation between $\mathcal{P}_i$ and $\mathcal{P}_j$. It can be described by the following formula (refer to Figure 2.2):

$$F_{ij} = \int_{A_i} \int_{A_j} \frac{\cos \phi_i \cos \phi_j}{\pi A_i r^2} dA_j dA_i \,. \tag{2.3}$$

[1]Since $\sum_{i=1,N} F_{ji} = 1$ for $j=1,...,N$, then $|-\rho_j \sum_{i=1,N,i\neq j} F_{ji}| < 1 - \rho_j F_{jj}$. Note that $0 \leq \rho_j < 1$ and normally $F_{jj} = 0$. See page 218 in [GoTo84].

Since the formula is important to the radiosity method to be developed, the following derivation for F_{ij} is necessary to understand the rest of the thesis.

First of all, consider the radiative exchange between differential elements. Note that (2.3) in fact describes the relations for radiative exchange between areas of finite size. Let dA_i and dA_j be two differential area elements on $\mathcal{P}_i$ and $\mathcal{P}_j$, that have their normals at angles ϕ_i and ϕ_j to the line of length r joining them, respectively. This notation and relation between the two elements are shown in Figure 2.2.

The total energy per unit time leaving dA_i and incident upon dA_j is

$$d^2P_{dA_i dA_j} = I_i dA_i \cos\phi_i d\omega_i , \tag{2.4}$$

where

$d\omega_i$ – the solid angle subtended by dA_j when viewed from dA_i,

I_i – the total intensity of surface dA_i, i.e., the total energy emitted by dA_i per unit time, per unit area dA_i projected normal to r, and per unit of solid angle.

The solid angle $d\omega_i$ is related to the projected area of dA_j and the distance r by the relation

$$d\omega_i = \frac{dA_j \cos\phi_j}{r^2} . \tag{2.5}$$

Substituting (2.5) into (2.4) gives the following:

$$d^2P_{dA_i dA_j} = \frac{I_i dA_i \cos\phi_i dA_j \cos\phi_j}{r^2} . \tag{2.6}$$

The total intensity is related to the total hemispherical emissive power by

$$I_i = \frac{\sigma T_i^4}{\pi} . \tag{2.7}$$

The total energy leaving $\mathcal{P}_i$ within the entire hemispherical solid angle over dA_i is expressed by $\sigma T_i^4 \, dA_i$ [1](See [SiHo78], pp.173 - 176). The notation T_i represents the temperature on

---

[1] σ is the Spefan-Baltzmann constant that is equal to 4.97×10^{-8}.

the element dA_i that is isothermal. Thus the fraction of the energy leaving dA_i and landing on dA_j, i.e., the form factor from dA_i to dA_j, is

$$dF_{d_i d_j} = \frac{d^2 P_{d_i d_j}}{\sigma T_i^4 dA_i}$$

$$= \frac{(\sigma T_i^4/\pi)(dA_i \cos\phi_i dA_j \cos\phi_j / r^2)}{\sigma T_i^4 dA_i} \tag{2.8}$$

$$= \frac{\cos\phi_i \cos\phi_j}{\pi r^2} dA_j \,. \tag{2.9}$$

By integrating (2.6) over the finite area A_j, the energy leaving dA_i and reaching all of A_j is

$$dP_{d_i j} = \int_{A_j} \frac{\sigma T_i^4 dA_i \cos\phi_i dA_j \cos\phi_j}{\pi r^2} \,. \tag{2.10}$$

Thus the form factor from the differential element dA_i to the finite area A_j is related by

$$F_{d_i j} = \frac{\int_{A_j} \sigma T_i^4 dA_i \cos\phi_i dA_j \cos\phi_j / \pi r^2}{\sigma T_i^4 dA_i}$$

$$= \int_{A_j} \frac{\cos\phi_i \cos\phi_j}{\pi r^2} dA_j \,. \tag{2.11}$$

If (2.6) is integrated over both A_i and A_j, then the result will be the energy leaving finite area A_i that reaches finite area A_j. Dividing the result by the total energy emitted by the entire area A_i results in the form factor from finite area A_i to finite area A_j as follows

$$F_{ij} = \frac{\int_{A_i} \int_{A_j} (\sigma T_i^4 \cos\phi_i \cos\phi_j / \pi r^2) dA_j dA_i}{\sigma T_i^4 A_i} \,, \tag{2.12}$$

then

$$F_{ij} = \int_{A_i} \int_{A_j} \frac{\cos\phi_i \cos\phi_j}{\pi A_i r^2} dA_j dA_i \,. \tag{2.13}$$

Several important points related to these equations need to be addressed because they become the guidelines for developing practical radiosity methods.

1. To obtain radiosities for different colors, (2.2) are solved independently three times, each for a color of red, green and blue with the corresponding color-sensitive E_j and ρ_j.

2. The above derivation assumes that all surfaces are perfectly diffuse. Without the assumption, the total energy leaving the source surface $\mathcal{P}_i$ would not be a constant; that is, the outgoing energy is different in different directions. In that case, the formula for form factors would be more complex than (2.13) [ImCo86, Kaji86]. In addition, another assumption simplifying the above derivation is that each finite area is isothermal at different temperatures. This assumption makes it possible that (2.13) can be derived from (2.12) by taking T_i^4 out of the integration and then canceling it from the fraction. This assumption is proved to be valid if each finite area is sufficiently small.

3. The formula for the form factor excludes the case in which surfaces are occluded by other surfaces. However, this situation happens in most object environments. To include this case, (2.13) needs to be modified by adding a function δ_{ij}.

$$F_{ij} = \int_{A_i} \int_{A_j} \frac{\cos \phi_i \cos \phi_j}{\pi A_i r^2} \delta_{ij} dA_j dA_i , \qquad (2.14)$$

where

$$\delta_{ij} = \begin{cases} 0 & \text{if } dA_i \text{ can not see } dA_j; \\ 1 & \text{otherwise.} \end{cases}$$

4. Notice that the radiosity, B_j, has a single value for the entire surface $\mathcal{P}_j$. This implies that the radiosity at each point on the surface should be equal. To meet this requirement, the area of $\mathcal{P}_j$ must be sufficiently small. This makes a radiosity method difficult to implement because surfaces are usually represented by 3-D polygons of certain sizes.

To solve this problem, a radiosity method often includes a preprocessing step that performs polygon subdivision.

5. From (2.13), it is easy to prove the following relationship:

$$A_i F_{ij} = A_j F_{ji}.$$

This equation is used to derive the radiosity system of simultaneous equations (2.1c).

2.2.2 Previous Work on Radiosity

A radiosity method is a computer implementation of applying the above theory to image rendering. Displaying an image needs intensity, or radiosity B_i ($i = 1, ...,N$), values for each pixel of the image. As indicated in (2.2), the central problem of obtaining radiosity turns out to be how to determine form factors F_{ij} ($i,j = 1,...,N$) efficiently. A number of papers have been published concerning this problem. Typical current work is described in this section.

PRIMARY METHOD

The formula for F_{ij} in (2.13) mainly consists of double area integrals. These computations make it difficult to calculate F_{ij} fast and accurately. Approximations to this integrals must be made. Goral et al [GoTo84] first introduced the radiosity method, and provided a primary approximate numerical method for calculating the form factor. Based on Stokes' theorem, this method converts the area integrals into double contour integrals:

$$F_{ij} = \frac{1}{2\pi A_i} \int_{C_i} \int_{C_j} [\ln(r)dx_i dx_j + \ln(r)dy_i dy_j + \ln(r)dz_i dz_j] \qquad (2.15)$$

and then uses two loops for C_j and C_i, respectively, to perform the integrals along the outlines of the areas. Note that r is the distance between (x_i, y_i, z_i) and (x_j, y_j, z_j). This method is very primitive. It is not feasible for a complicated scene and not valid for a scene with occluded surfaces.

HEMI-CUBE

Calculating logarithms involved in (2.15) is expensive, and it is not practical to exclude scenes with occluded surfaces. Cohen et al. made further developments based on Goral's work, and proposed a feasible method for finding approximate form factors considering occluded surfaces [CoGr85]. This approach is based on the Nusselt analog [SiHo78]: the form-factor between two patches, one called the receiving patch, and the other the emitting patch[1], is obtained by two processes of projection. First, the receiving patch is projected onto the unit hemisphere surrounding the emitting surface center. The projection on the hemisphere is then orthographically projected down onto the base of the hemisphere. The form-factor from the emitting surface to the receiving surface is then equal to the area covered by the second projection divided by π (refer to Figure 4.4). A hemisphere is difficult to manipulate. Cohen uses a hemi-cube rather than a hemisphere as the planes onto which patches are projected. This method provides the base on which our **FBRM** was developed. It will be examined in detail in Section 4.2.

SUBSTRUCTURING

To speed up calculating form factors and the radiosity solution, Cohen [CoGr86] improved the hemi-cube method based on the following observation: radiosity methods based on coarse meshing are fast but not accurate; fine-meshing methods are accurate but slow. Therefore, a trade-off is necessary. Global radiosity is first calculated according to coarse meshing, and then local radiosity is obtained by using fine patch meshing. The basic idea is to use the concept of *substructuring*: surfaces are subdivided into patches each of which has constant material properties and each patch is also subdivided into elements, each having constant radiosity. After this, element-to-patch form-factors are calculated. Using the element-to-patch form-factors, the radiosity of each element can be obtained by

---

[1]The meaning of the receiving patch and the emitting patch will become clear through discussion in Chapter 4.

applying the coarse-patch global radiosities previously obtained. The depth of element subdivision within a patch is dependent on radiosity gradients across the patch. A high gradient patch requires more subdivision. Cohen suggested an adaptive subdivision technique by which the level of subdivision can be controlled dynamically.

SIMPLE METHOD FOR NON-DIFFUSE

Immel et al. made efforts in investigating a radiosity method for non-diffuse environments [ImCo86]. With their approach, the restriction that each surface is perfectly diffuse is removed from the previous methods. In this scheme, radiant energies in all directions are calculated independently and differently. To compute the directional radiosity, three kinds of discretizations need to be performed: discretization of surfaces, discretization of visible spectrum, and discretization of solid angles. With these discretizations, a formula similar to the form-factor calculation in a diffuse environment is found. A set of simultaneous equations are solved to obtain directional radiosity on each patch in all the discretized directions.

TWO-PASS FOR NON-DIFFUSE

In the SIGGRAPH'87 conference, Wallace et al. proposed a two-pass method for non-diffuse environments [WaCo87]. In the first pass, the view-independent technique, the hemi-cube method is employed to calculate the diffuse component of global illumination. In the second pass, a view-dependent technique mainly represented by ray-tracing is used to determine the specular and transmitted reflections.

PROGRESSIVE REFINEMENT

In recent years, researchers paid more attention to the hemi-cube method for an environment with only diffuse surfaces. The progressive refinement radiosity proposed by Cohen [CoCh88] utilizes the hemi-cube method to interactively drive the rendering process from coarse images that estimate radiosity to the final fine images. The refinement process

starts with energy from bright surfaces like light sources to produce initial images. By user interaction, the image is gracefully and continuously refined by including light from the secondary light sources. The refinement procedure continues until a satisfactory image is obtained. The major difference between this approach and conventional radiosity techniques is the ability to compute form-factors on-the-fly, that eliminates the huge memory requirement for form-factors and the unnecessary computation.

RAY-TRACE BASED PROGRESSIVE

In SIGGRAPH'89, a couple of papers were concerned with progressive approaches, each with improvements at different levels. Wallace [WaEl89] presented a progressive approach that utilizes the ray tracing technique instead of the hemi-cube to determine form factors. Form factors are calculated by emanating rays from vertices of each polygon surface to the sample points on the primary or secondary sources. The advantage of this method is that aliasing problems in computing form-factors can be reduced.

Another paper in SIGGRAPH'89 [BaRu89] introduced an analytical method that was also based on the progressive approach. This method is a hybrid combination of analytical and numerical techniques for form factor computation. It computes form factors analytically when the hemi-cube assumptions are violated and numerically computes form-factors otherwise.

These two papers in SIGGRAPH'89 attempt mainly to reduce restrictions imposed on the traditional radiosity methods. Their methodology opens new directions for radiosity researchers to pursue more advanced methods.

RECENT RESEARCH ON RADIOSITY

Recently, a few papers about radiosity have been published in SIGGRAPH' 90. These papers tackled radiosity in different aspects, but based on the existing techniques. Chen et al. proposed a radiosity solution for bump texture generation in [ChWu90]. Their

approach used Blinn's "bumpy mapping" to calculate a *perturbed radiosity map*, which can be used for later texture generation. The article in [Chen90] is mainly concerned with recomputing images efficiently when object attributes have been changed incrementally. The progressive method was modified to achieve the incremental radiosity. Heckbert in [Heck90] proposed another hybrid method to handle radiosity in a non-diffuse environment including texture. Campbell et al. in [CaFu90] focused on the mesh technique for surface elements that are required by any radiosity method. A *BSP* tree based shadow algorithm is used to perform polygon subdivision and visibility for form-factors.

More recently, several research papers have been published to improve quality of images generated by the radiosity method. Heckbert addressed the problems of discontinuities caused by shadows in his Ph.D. thesis [Heck91]. Tampieri et al. in [TaLi91] proposed a progressive refinement method to eliminate the assumption that the radiosity of the shooting patch is constant. (Tampieri also did research on parallel implementations of radiosity algorithms in his M.Sc. thesis [Tamp89].) Other recent work on the radiosity methods can be found in [LiTa91] and [HaSa91].

2.2.3 Implementation

In general, a standard radiosity method for a diffuse environment consists of four steps:

- subdivide surfaces into patches, each having a constant radiosity;
- calculate the form-factors from each patch to the remaining patches;
- solve a set of simultaneous equations for radiosity;
- render the image using an appropriate rendering algorithm.

The first step is necessary for guaranteeing the assumption for (2.13) that each finite area $\mathcal{P}_j$ has a constant radiosity. Strictly speaking, the intensities at any two points on a physical surface are different. But the difference is not noticeable if the two points are close. In other words, if a surface is sufficiently small, the intensity is regarded as a

constant. The first step in a radiosity method attempts to break a big polygon into patches[1] that are as small as necessary. However, a large number of small patches will have a computational impact on the second step. Thus, how to subdivide polygons depends on the material properties of these polygons and their relation with illumination sources. For example, a polygon may be subdivided into fewer and bigger patches if the radiosity across the polygon surface varies slightly.

Usually, the subdivision operation requires a certain amount of human interaction and is difficult to be completed wth sufficient accuracy. Furthermore, a polygon is subdivided into equally-sized rectangular patches so that radiosity errors otherwise caused by odd-shaped patches can be reduced. In order to mesh polygons more accurately in terms of radiosity, fine subdivision is needed in the surface areas with complex intensity gradients; in other words, these surfaces are divided into patches smaller than other surfaces. Strictly speaking, the subdivided patches can never meet the requirement of constant radiosity in most cases, but a good approximation to polygon subdivision suffices for radiosity image generation.

The second step is the kernel of the radiosity method. Since this step involves computing the visibility of patches, it is the most expensive part. Its efficiency controls the overall performance of the method. To accomplish the visibility task, a hidden surface removal algorithm must be included. In the hemi-cube, the z-buffer algorithm is used for this purpose because of its simplicity. Chapter 4 will address how the hemi-cube implements this step.

The last two steps are easy to implement. Any numerical method for solving a set of simultaneous equations can be used to obtain radiosity solutions. But the Gauss-Seidel method is more favorable for the radiosity equations because (2.2) is diagonally dominant.

---

[1] Throughout this thesis, for convention, a patch always refers to a small polygon that has a constant radiosity.

Once radiosity values for each patch have been obtained, rendering images becomes a trivial problem. Any rendering procedure can be used for this purpose.

Chapter 3

3-D Subdivision of Polygons

To guarantee accuracy of radiosity solutions to (2.2), each polygon must be sufficiently small so that its radiosity is invariant. In general, polygons appear in large sizes and the radiosity across their surfaces is not constant. Thus, subdividing polygons into small patches should be performed wherever necessary. Traditionally, a polygon is meshed into a number of small, sometimes regular, patches along the polygon surface. This chapter presents a new subdivision scheme that uses an octree technique. Since the polygon subdivision is performed in 3-D space, it is called 3-D subdivision to distinguish it from the traditional approach.

3.1 Introduction

The requirement that each polygon has a constant radiosity makes it difficult to implement the radiosity method. In general, it is impossible to obtain a polygon having constant radiosity in reality unless the polygon is a point. A small polygon is, however, considered to have an approximately constant radiosity. A meshing technique, therefore, needs to be employed to subdivide polygons into smaller polygons. The question is: how small should an initial polygon be subdivided? This is an interesting and difficult problem.

Traditional subdivision methods mesh a polygon into regular rectangles or triangles. Meshing is always performed on the planes of polygon surfaces. Figure 3.1 shows a result of the subdivision where the resulting polygons have the same size and shape except those close to the boundaries. Regularity of meshing is likely to reduce

aliasing occurring in calculating form-factors and makes the subdivision easy to implement. Odd-shaped patches will partially occlude many pixels of the hemi-cube. For each partially occluded pixel, the algorithm must arbitrarily decide whether or not that pixel contributes to the projected patch's form-factor. On the other hand, regular patches will cover the hemi-cube in a regular pattern. In general, partially covered pixels could be reduced. It is, however, difficult to control the level of subdivision and automate an adaptive subdivision.

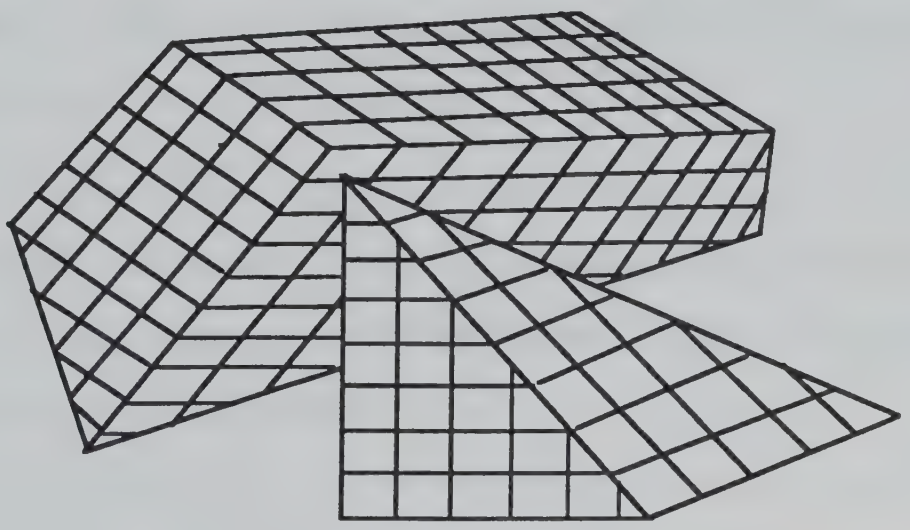


Figure 3.1. Regular mesh for polygons. Polygon surfaces are meshed into regular shapes by traditional subdivision techniques.

The level of subdivision is important to both quality and efficiency of the radiosity algorthm. A lower level subdivision gives a coarse meshing with larger patches. The larger the patches, the more inaccurate the form-factors and thus the quality of images becomes poorer. On the other hand, a higher level subdivision produces an excessive number of patches that significantly increases the execution time of the radiosity.

As a result, the basic principle is that subdivision should be performed in a way adaptive to the distribution of intensity gradients. In other words, places with higher intensity gradients should have a finer meshing than places with lower intensity gradients.

The traditional regular meshing prevents such an adaptive subdivision since it lacks knowledge of intensity gradients on the polygon surfaces being subdivided. Thus traditionally, the solution to an adaptive subdivision uses post-processing to manually divide high-intensity-gradient polygons into fine patches.

Manual subdivision can yield fairly good meshing, but it depends on the operator's understanding of illumination and requires a sophisticated interactive system to input subdivision specifications. In addition, it is easy to make mistakes of judgement on intensity gradients. Such errors could cause serious artifacts on the final images. As a result, automatic adaptive subdivision is desired.

The quadtree method that is usually used to represent a 2-D image can be a candidate for adaptive subdivision. Unlike regular meshing, this method controls the level of subdivision at will according to pre-defined criteria. Figure 3.2 demonstrates the partitioning of a polygon using a quadtree.

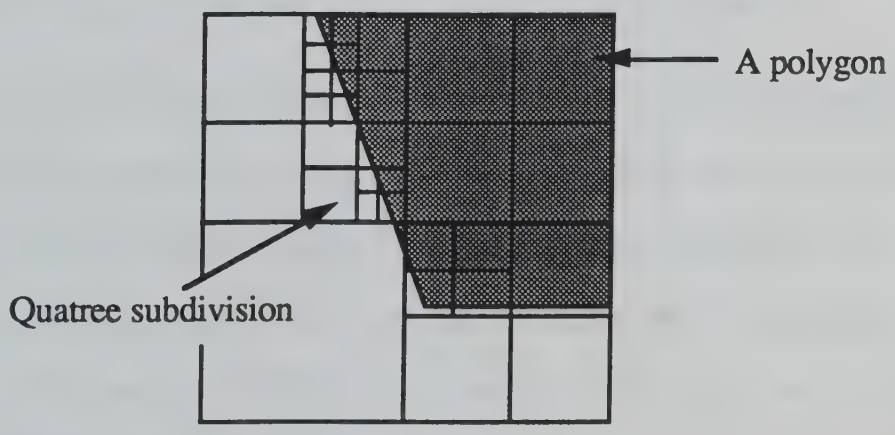


Figure 3.2. Polygon subdivision using a quadtree.

A quadtree is used n a 2-D space. Its extension to 3-D is the *octree*. Based on the characteristics of form-factor calculation, the octree rather than the quadtree is adopted to perform the polygon subdivision in the **FBRM**. It is called 3-D subdivision throughout

this thesis. To achieve the goal of automatic adaptive subdivision, all polygons in a model are preprocessed before 3-D subdivision is applied. The preprocessing is discussed first in the next section.

In order to make the subdivision algorithms described in the subsequent sections work properly, especially for the sake of the implementation of **FBRM**, three assumptions have been made. First, it is assumed that all surfaces of objects in a model must be polygons. Curved surfaces have to be meshed into polygons by any meshing technique, and will not be described in the thesis. In the implementation of **FBRM**, curved surfaces are not considered.

The second assumption is that all polygons under consideration must be non-penetrating and simple. A simple solution to penetrating polygons is to manually split them into separate polygons along the intersections. Generally speaking, polygons with holes are not allowed. This is mainly because the clipping algorithm used in **FBRM** is unable to clip this type of polygons correctly. Again, polygons containing holes can be broken into several simple polygons. Note that concave polygons are permitted.

The third assumption is that different polygons from the original model either belong to different objects or form different surfaces of the same object before they are processed. In other words, the original polygon surfaces do not have any "cracks". As will be seen, cracked polygons may create an excessive number of patches that have an impact on the efficiency of **FBRM**. This assumption may be removed if "cracked" polygons can not be avoided.

3.2 Preprocessing - Shadow Boundary Computation

The objective of preprocessing is to identify the areas that have high intensity gradients across polygon surfaces. In general, complex intensity gradients occur around the boundaries of shadows and the edges shared by several polygons. The edges joined by

multiple polygons are already identified by themselves. Thus shadows are the only concern in this preprocessing step.

To determine the shadow boundary, the easiest way is to split a shadowed polygon into lit subpolygons and unlit subpolygons. Accurate shadow determination requires a complicated and sophisticated algorithm. In particular, it is very difficult to determine shadows in an environment involving area light sources. This section only gives a proposal for obtaining the shadow information by using Atherton-Weiler-Greenberg shadow algorithm. Although Atherton's shadow algorithm yields only a list of lit polygons, the shadow boundaries can be calculated by testing the lit polygons against the original polygons in the modelling database.

3.2.1 Notations for Polygons

Three notations are used to express different polygons involved in this thesis. The original polygons that constitute faces of objects are called "*polygons*", denoted by ξ_i ($i = 1, ..., L$). The set of ξ_i is expressed by $\Xi = \{\xi_1, ..., \xi_L\}$ representing the original modelling database.

Polygons obtained by the preprocessing are called "*lit-polygons*" expressed by χ_i ($i = 1, ..., M$). The set of all the polygons is expressed by $\Xi\text{-}lit = \{\chi_1, ..., \chi_M\}$. The relation between χ_i and its parent polygon is important to the 3-D subdivision, which will become clear later. Thus, j_i is used to indicate the index of χ_i's parent polygon.

By using the lit information provided in $\Xi\text{-}lit$, the 3-D subdivision to be stated in the next section subdivides polygons in Ξ into small polygons such that they have constant radiosity. Such small polygons are called "*patches*", expressed by $\mathcal{P}_i$ ($i = 1, ..., N$). The collection of all patches are denoted by $P = \{\mathcal{P}_1, ..., \mathcal{P}_N\}$. Figure 3.3 shows the relationship among these polygons.

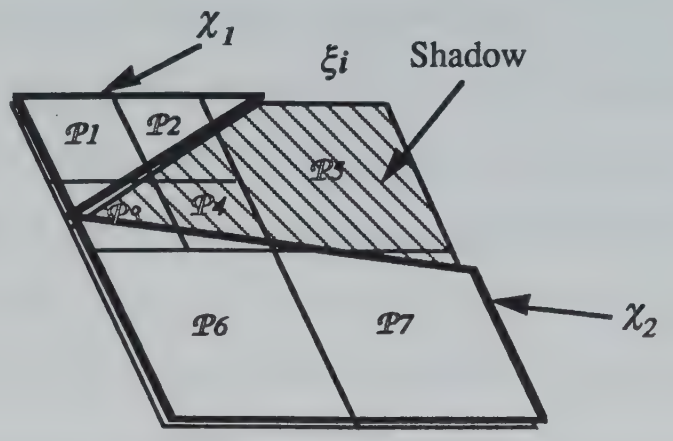


Figure 3.3. Relationship among *polygon*, *lit polygon* and *patch*.

3.2.2 Determination of Shadow Boundaries

As mentioned above, Atherton-Weiler-Greenberg's two-pass object-precision shadow algorithm [AtWe78] is employed to determine shadow boundaries. In this section, Atherton's shadow algorithm is described first, and how to determine shadow boundaries using the list of lit polygons is then discussed.

Atherton-Weiler-Greenberg's first pass, using the Weiler-Atherton hidden surface removal algorithm, outputs a subset of the original modelling database that only contains those surfaces visible from the viewpoint of a point light. The lit polygons are then transformed back into the modelling coordinates and merged with the original database. The second pass performs hidden-surface removal and the shading operation on the merged database from the actual viewpoint of an observer. Since the shading operation is of no interest to us, the following discussion concentrates on the first pass, particularly, the Weiler-Atherton visible surface algorithm that is used to output the lit polygons in Atherton's shadow algorithm.

The Weiler-Atherton visible surface algorithm first sorts polygons in the original database Ξ on their nearest z coordinate. Suppose that Ξ-*list*= $\{\xi_1, ..., \xi_L\}$ is the result of the sort where ξ_i is closer to the viewpoint than ξ_j if $i<j$. The first polygon ξ_1 in Ξ-*list* is then used to clip all polygons in Ξ-*list* including ξ_1 into two lists: *in-list* and *out-list* containing the clipped pieces inside and outside the clip polygon ξ_1, respectively. The clipping operation takes three steps. First of all, each polygon ξ_i is projected onto the image plane with respect to the viewpoint of the light. ξ_i is then clipped against ξ_1 on the image plane. Finally, the clipped pieces are transformed back to the modelling coordinates.

All polygons in the *in-list* that are behind ξ_1 are discarded since they are blocked by ξ_1. If the *in-list* contains only one polygon ξ_1, ξ_1 is removed and placed in the Ξ-*lit* that contains the lit polygons. If a polygon in the *in-list* is in front of ξ_1, this implies that the early *z*-based sort does not give a correct priority order for Ξ-*list*. The misplaced polygon needs to be adjusted by the following recursive steps.

The algorithm uses a stack to handle penetrating polygons. When a misplaced polygon is found, the current clip polygon is pushed into the stack. Then using each of such misplaced polygons as the clip polygon, the shadow algorithm is recursively called to clip the pieces on the *in-list* against the clip polygon. If a polygon is found to be in front of the current clip polygon, it is searched for in the stack. If it is on the stack, then no more recursion is needed. This is because all polygons inside and behind that found polygon have already been removed. When the recursive process is over, the polygons in the *in-list* are removed and placed in Ξ-*lit*. One of reasons for using the stack is to guarantee this algorithm to terminate when those cases of cyclic overlap happen[1].

---

[1]In our implementation of **FBRM**, actually, it is assumed that **FBRM**only handles non-penetrating polygons.

After the *in-list* is empty, the Ξ*-list* is replaced by the *out-list*, and the algorithm then continues to process the polygons in the Ξ*-list* (i.e. the *out-list*) recursively until no more elements are in the *out-list*. Finally, the list Ξ*-lit* is output to the 3-D subdivision.

Pseudocode for the Weiler-Atherton algorithm is shown in **ALGORITHM 1**.

ALGORITHM 1 (Weiler-Atherton Algorithm)

Several list-operation functions are used in the algorithm. **CAR**(*a-list*) returns the first element in the list, **CDR**(*a-list*) returns the list with the first element removed, **INSERT**(*an-element*, *a-list*) places the element at the first place of the list, **APPEND**(*an-element*, *a-list*) places the element at the last place of the list, and **NUMBER**(*a-list*) returns the number of elements in the list.

```
PolygonList Ξ-list ⇐ the original modelling database Ξ;
PolygonList Ξ-lit ⇐ empty;
Stack stack;

ShadowClipper( )
{

sort Ξ-list by decreasing value of maximum z;
stack ⇐ empty;

/* process each polygon remaining in Ξ-list */
while( Ξ-list is not empty )
        SCSubdivide(CAR(Ξ-list), Ξ-list);

} /* ShadowClipper */

SCSubdivide( Polygon ξclip, PolygonList plist )
{
PolygonList in-list, out-list;
Polygon ξ;

in-list ⇐ empty;
out-list ⇐ empty;

/* Clip polygons in plist into two lists: in-list and out-list */
while(plist ≠ empty) {
        ξ = CAR(plist);
```

```
        clip ξ to parent of ξclip;
        place pieces inside ξclip in in-list and pieces outside ξclip in out-list;
        plist ⇐ CDR(plist);
}

/* Remove shadowed polygons from in-list */
Num = NUMBER(in-list);
for( i=1; i< Num; i++) {
        ξ = CAR(in-list);
        in-list ⇐ CDR(in-list);
        if( ξ is in front of ξclip)
                APPEND(ξ, in-list);
}

/* Process incorrectly ordered polygons recursively */
for( i=1; i<=NUMBER(in-list); i++) {
        ξ = CAR(in-list);
        if( ξ is on stack or a part of ξclip ) continue;
        PUSH(ξclip, stack);
        SCSubdivide(ξ, &in-list);
        POP(stack);
}

/* Output remaining polygons in in-list to Ξ-lit */
for( i=1; i<=NUMBER(in-list); i++) {
        ξ = CAR(in-list);
        INSERT( ξ, Ξ-lit);
}

/* Replace Ξ-list by out-list to continue ShadowClipper on the remaining polygons */
Ξ-list ⇐ out-list;
} /* SCSubdivide */
```

In the above algorithm, clipping operations are actually not described. The reason for this is the requirement for a powerful clipping algorithm as indicated in [FoDa90, Page 689]. Although we made an assumption that no polygon contains holes, the Weiler-Atherton algorithm may fail if only a simple clipping method is used. Clipping polygons against shadows will, perhaps quite often, produce lit polygons that have holes (polygons with holes can be split into siimple polygons). Therefore, this approach has to require a powerful clipper, such as the Weiler polygon algorithm described in Section 19.1.4 of [FoDa90], that can clip one polygon with holes against another arbitrary polygon. Because

of its complexity, the description of the algorithm is omitted. It is always assumed that a powerful clipping algorithm is used in the Weiler-Atherton algorithm.

Note that clipping operations are performed with a copy of the original database, referring to the first line of the algorithm. Each clipped piece must point back to its "*parent polygon*" from which it is originally derived. With the back-to-parent pointer, when each piece is used as the clip polygon, all polygons are actually clipped against its parent polygon since it is more efficient than clipping polygons against the clip polygon itself.

In addition, the back-to-parent pointer is used by the 3-D subdivision when a fine subdivision is performed. The 3-D subdivision operates on both Ξ and Ξ-*lit* at the same time; that is, the union of Ξ and Ξ-*lit* is regarded to be one single database with which the 3-D subdivision divides polygons into patches. For each patch $\mathcal{P}_{lit}$ derived from Ξ-*lit*, a test against the patch $\mathcal{P}$, that has the same parent as $\mathcal{P}_{lit}$, but is derived from Ξ, is performed in order to determine the shadow boundary. If the two patches have the same vertices, no special operation is performed because no shadow boundary crosses patch $\mathcal{P}$. If not, this implies that a shadow boundary is cast on $\mathcal{P}$. In this case, therefore, $\mathcal{P}$ needs to be subdivided into smaller patches. In Section 3.4, we will discuss how to perform the test through the *criterion vector* in more detail.

Radiosity normally models an environment with polygonal light sources instead of point light sources. To compute shadows resulting from such light sources, Atherton's shadow algorithm needs to be executed at each vertex of the polygonal light source. Each execution of the algorithm produces a list of lit polygons output to the 3-D subdivision. The 3-D subdivision, therefore, has several lists of Ξ-*lit*. To determine shadow boundaries cast by each light vertex, the above test is performed for each Ξ-*lit* independently. Figure 3.4 demonstrates the shadows cast from a triangular light source. For each edge of shaded polygons in Figure 3.4, which is also one edge of a lit polygon, an extensive subdivision

will be applied by the 3-D subdivision; therefore, areas close to shadow boundaries are eventually divided into small patches.

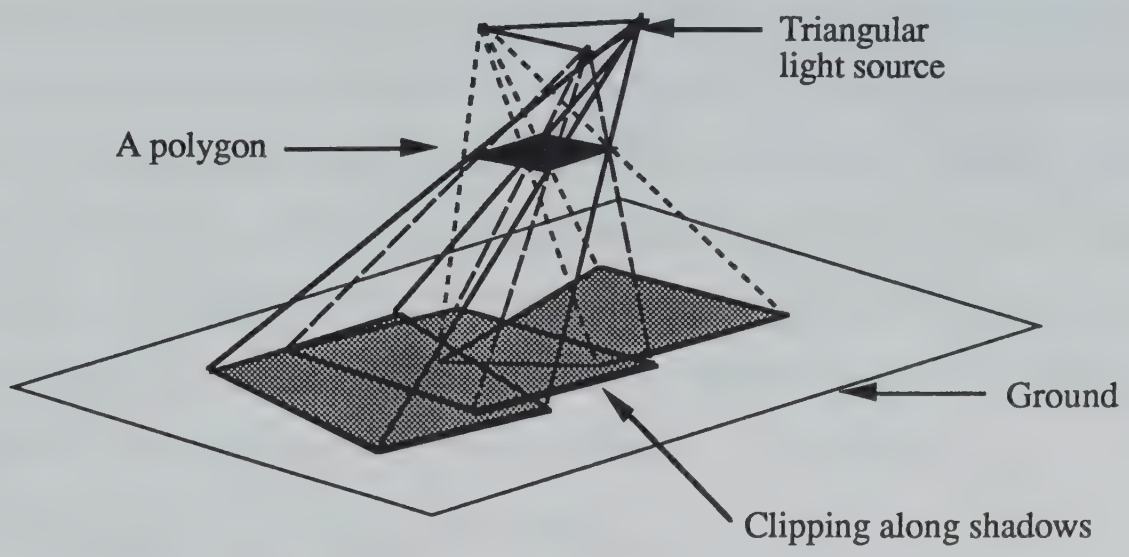


Figure 3.4. An approximation to the shadow's umbra and penumbra. Assume that shadows are cast from each vertex of the polygonal light source. For each vertex, the polygonal shadow is identified by the lit polygon(s) surrounding the shadow. The lit polygons are output and will be used in the 3-D subdivision.

Generally, a shadow includes two parts, *umbra* and *penumbra*, if an area light is used. Our algorithm does not distinguish them, but implicitly considers the shadow's umbra and penumbra. In other words, the areas close to the edges of both umbra and penumbra are divided into more patches because these edges are also part of boundaries of lit polygons in some *Ξ-lit*. An excessive number of patches might be generated close to those parts of a lit polygon's boundaries that are on neither the umbra's and nor the penumbra's edges. As pointed out, the shadow information is not used to compute accurate shadows but to guide the 3-D subdivision to do an adaptive subdivision. The unnecessarily generated patches could have some impact on the speed; however, in some sense, these areas of a lit polygon's boundaries may deserve more patches because intensity around those areas is not as simple as on a non-shadowed area.

3.3 Octree for 3-D Subdivision

Space partitioning is usually regarded as an effective technique to improve image generation. An octree is one such technique. The fast ray tracing algorithm proposed by Glassner 1984 [Gl84] was based on an octree technique. Many papers dealing with the octree subdivision have been published [FuTa86, Ka85, SaWe88]. However, there are presently no papers that tackle radiosity using octree techniques. Although Cohen et al. and Samet-Webber mentioned how to take advantage of hierarchical structures to improve efficiency of radiosity [CoGr86, SaWe88], the detail of implementation was not described. This section is devoted to discussion of an octree technique used in the **FBRM**. In the next chapter, how it speeds up the radiosity method will be discussed.

3.3.1 Numbering Scheme for Octree

Without loss of generality, it can be assumed that all polygons in a 3-D environment are enclosed in a unit cube whose edges are parallel to the three axes of the world coordinate system. For a non-unit cube, the maximum length at each axis is multiplied to the equations for dividing planes used in the unit cube.

An octree method subdivides the cubic space that encloses all the objects under consideration into 8 equal-sized small cubes; then each sub-cube is recursively subdivided into smaller 8 sub-sub-cubes. This process continues with each subsequent cube until the given criterion is met. When the entire process stops, an octree is obtained, in which each node has at most 8 children. The octree relates all cubes or sub-cubes in a hierarchical structure. For descriptive convenience, the leaf cubes in the octree are called octants, and the others are called cubes. To access octants through the octree, a naming scheme is needed. Gargantini proposed a numbering scheme [Ga82], in which 0 through 7 are used to name the nodes. Although Glassner had a better numbering system [Gl84], this paper mainly refers to Gargantini's method since its features are more suitable for the **FBRM**.

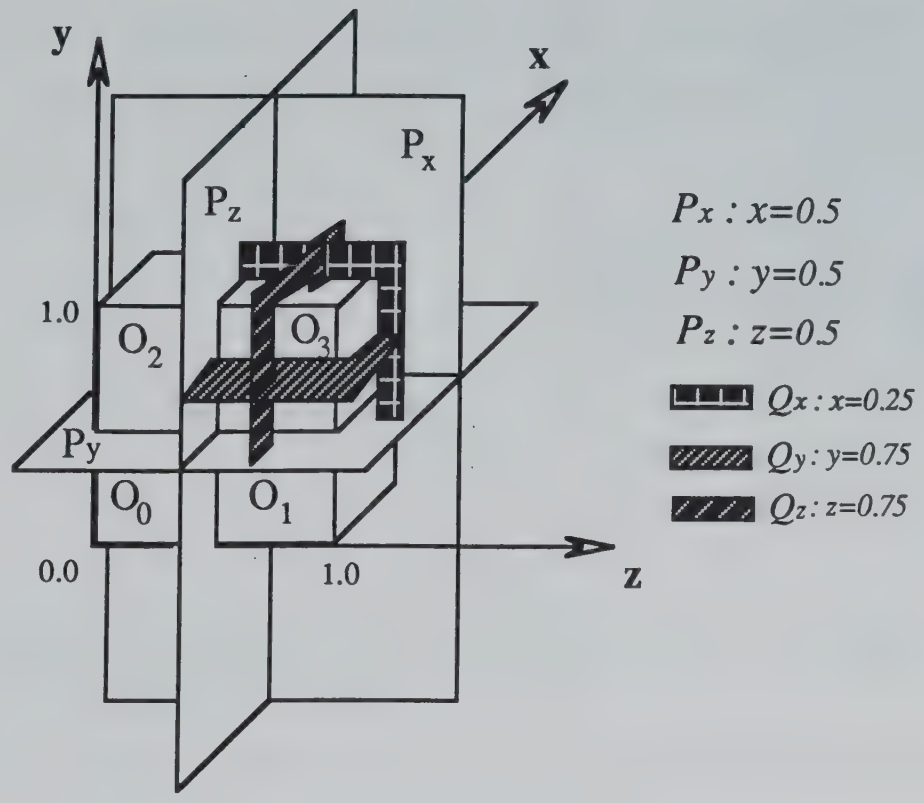


Figure 3.5. Cube subdivision. A cube is subdivided into 8 smaller sub-cubes by P_x, P_y and P_z. The sub-cube O_3 is divided again by Q_x, Q_y and Q_z.

Starting with the initial unit cube, the whole environment is subdivided into 8 sub-cubes by 3 planes P_x ($x=0.5$), P_y ($y=0.5$) and P_z ($z=0.5$), as shown in Figure 3.5. An octal number $\delta = \delta_x\delta_y\delta_z$ is assigned to each sub-cube where each of δ_x, δ_y, and δ_z is either, 0 or 1. Let (x, y, z) represent the corner of each sub-cube that is the closest to the origin. Suppose that (x_m,y_m,z_m) is the center of the initial unit cube. The code for each sub-cube is obtained by

$$\delta_x = \begin{cases} 0 \text{ if } x<x_m, \\ 1 \text{ otherwise,} \end{cases} \quad \delta_y = \begin{cases} 0 \text{ if } y<y_m, \\ 1 \text{ otherwise,} \end{cases} \quad \delta_z = \begin{cases} 0 \text{ if } z<z_m, \\ 1 \text{ otherwise.} \end{cases} \tag{3.1}$$

Thus, 8 octal numbers from 0 through 7 uniquely identify the 8 sub-cubes. After the initial cube is subdivided into 8 sub-cubes, each sub-cube is tested against the following three cases.

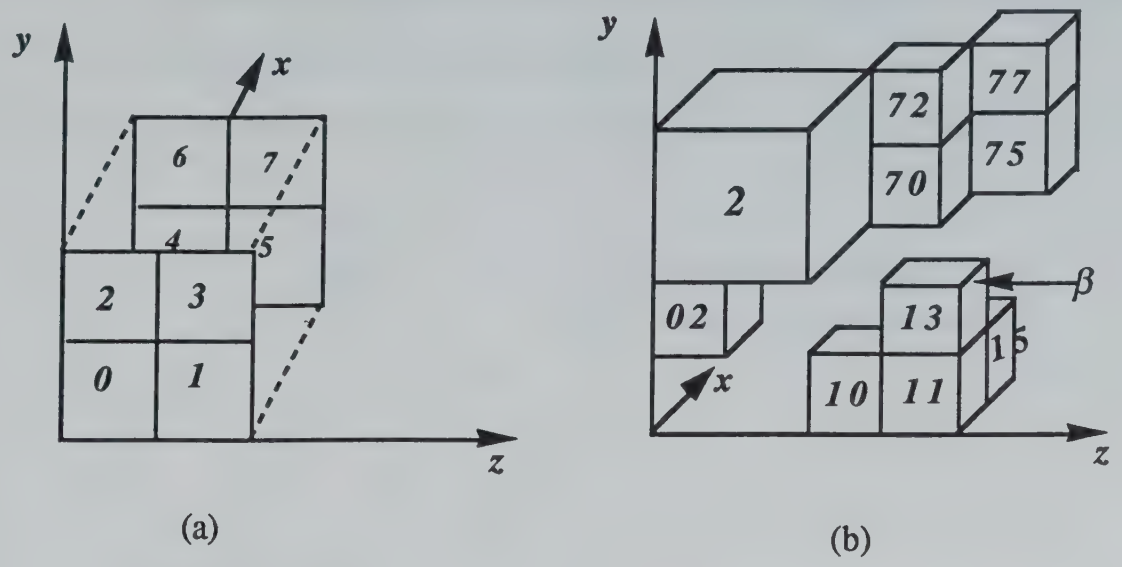


Figure 3.6. The numbering scheme for the octree. (a) Assignment of numbers to each sub-cube at any level of the octree. (b) An example of using this numbering scheme.

CASE 1 If the sub-cube is *empty*, the sub-cube is discarded. In the 3-D subdivision, *empty* means that no polygon in the model lies inside the sub-cube.

CASE 2 If the sub-cube is *solid*, meaning that it, as well as polygons inside it, meet a given criterion, no further processing of the sub-cube need be done. Thus an octant is obtained.

CASE 3 If a sub-cube is *partially solid*, implying that it does not meet the criteria, another set of planes are applied to subdivide the sub-cube into 8 smaller sub-sub-cubes. For example, the planes: Q_X: x=0.25, Q_Y: y=0.75, Q_Z: z=0.75, are used to divide O_3 in Figure 3.5. Again, 8 octal digits are obtained according to (3.1). A two-place octal number is used to represent each sub-sub-cube where the first place is the octal for its parent cube, and the second is one of the octal digits obtained by (3.1). An example of the numbering is shown in Figure 3.6. The sub-sub-

cube β is assigned to $(13)_8$ where 1 is the code for its parent sub-cube and 3 is the newly-derived octal.

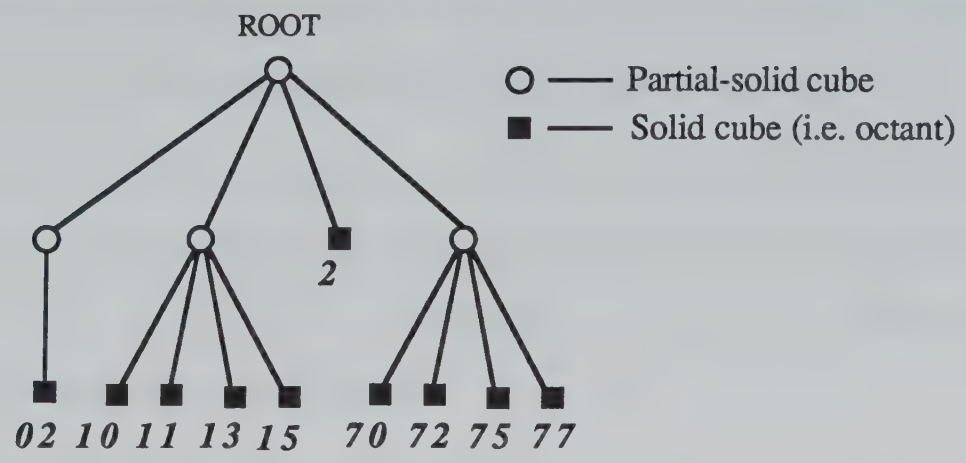


Figure 3.7. An octree corresponding to the example in Figure 3.6 (b).

The subdivision continues to process each subsequent *partially solid* cube. Eventually, the environment space is divided into small cubes in various sizes, each of which is identified by an octal number. The process of the subdivision can be represented by a tree structure as shown in Figure 3.7 where a solid square stands for an octant, and a hollow circle represents a partial-solid cube.

With this numbering scheme, a problem has been found: the same octal number could represent several cubes or octants. For example, in Figure 3.6 (b), the octal 2 could represent both the left-upper cube and the cube marked with "02". In order to distinguish these ambiguous cubes or octants, the code for each cube or octant is extended from a number to a vector (k, l) where k is the octal number, and l indicates the level of the cube in the octree. Thus (2, 1) represents the left-upper corner, while (2, 2) stands for the "02" cube. For descriptive purpose, (2, 2) is sometimes expressed by the octal 02, while (2,1) is expressed by 2.

Using explicit pointers to represent an octree, as done in Figure 3.7, is straightforward and easy to implement, but not economic in terms of memory. To save memory, Gargantini [Garg82] proposed a *linear octree* notation in which the leaf nodes of the octree are expressed by a sequence of codes or numbers that implies a tree structure. It is found that the linear octree is particularly useful for our problem.

The linear notation to represent our octree is a sequence of vectors (k, l) associated with octants. Thus the linear representation for the octree shown in Figure 3.7 is (2, 2) (10, 2) (11, 2) (13, 2) (15, 2) (2, 1) (70, 2) (72, 2) (75, 2) (77, 2). More discussion on the linear octree will appear in Chapter 4.

An important advantage of this numbering scheme is that it is easy to locate any octant simply by mapping its corresponding code back into the nearest corner coordinate (x, y, z), referring to (3.1). The arrangement of the coordinate system shown in Figure 3.6 is designed for the purpose of calculating octant locations easily.

3.3.2 Clipping of Polygons

Constructing an octree is not the whole story; using the octree to subdivide polygons into patches is the main purpose. To include the polygon clipping operation, the notation for a cube needs to be extended again. We use the letter O_k to represent a cube where k is an index to distinguish different cubes, which happens to be equal to the octal number assigned to the cube. O_k consists of six elements as follows: $\{k, l, xm, ym, zm, \Xi_k\}$ where k is an octal number, l is the level of the cube appearing in the octree, (xm, ym, zm) is the center coordinate of the cube, and Ξ_k is a list of those polygons ξ_k that lie entirely in O_k.

As discussed above, for each vertex of each polygonal light source, the preprocessing step subdivides polygons along shadow boundaries into lit polygons, outputting a list: *Ξ-lit*

$= \{\chi_1, ..., \chi_M\}$[1]. All these Ξ-*lit*'s and the original database Ξ are fed into the 3-D subdivision step where operations on any of Ξ-*lit*'s and Ξ are performed independently of each other. In other words, the 3-D subdivision will subdivide polygons in each of these databases independently though the same polygons possibly belong to different databases. The subdivision on a Ξ-*lit* is used only for the criterion test in order to perform an extensive subdivision wherever necessary. The criterion test will not be discussed until Section 3.4. In the following discussion, therefore, we assume that the database fed to the 3-D subdivision is Ξ. Suppose that the initial unit cube encloses all the polygons in Ξ. Thus, the initial unit cube can be expressed by O_0= {0, 0, 0.5,0.5,0.5,Ξ}.

As a *partial-solid* cube O_k is subdivided into 8 sub-cubes by the dividing planes, for example, P_x, P_y and P_z in Figure 3.5, any polygon ξ_k in Ξ_k that crosses the dividing planes is simultaneously clipped by the planes. The polygon subdivision goes on as subsequent *partial-solid* cubes are subdivided again. Once the octree is completed, each polygon in Ξ is eventually divided into a number of small polygons that lie entirely in one of the leaf octants. If the criteria for subdivision termination are chosen carefully so that each octant is sufficiently small, each of these small polygons will be guaranteed to have a constant radiosity. Therefore, the union of all polygons in the octants yields the list of patches: $P = \{\mathcal{P}_1, ..., \mathcal{P}_H\}$.

3.3.3 3-D Subdivision Algorithm

The previous subsections have given a basic idea of how the 3-D subdivision works. Basically, the 3-D subdivision accomplishes two tasks: constructing a linear octree and clipping polygons into patches. We are now in the position to describe in detail the 3-D subdivision algorithm.

---

[1]Each element in this list should point back to its parent, expressed by $\chi_i^{j_i}$. For descriptive convenience, the tedious superscript is omitted. But it should be kept in mind that the back-to-parent pointer is always there, and it will also be associated with the patches derived by the 3-D subdivision.

There are three lists of cubes used in the algorithm. ***FOL*** is a list of the octants, called *Final Octant List.* ***AOL*** (*Active Octant List*) is a *stack* containing partial-solid cubes. ***TOL*** is a temporary list containing intermediate results by subdividing a cube. The ultimate goal of the 3-D subdivision is to generate the ***FOL*** with its elements sorted in the natural order of leaves in the octree, that is, octants having the same parent appear in a neighborhood in the ***FOL***.

At the beginning of the algorithm, ***AOL*** is initialized by O_0. Each element O_k in ***AOL*** is then split into 8 sub-cubes by three processes: *process x*, *process y* and *process z*. These three processes work as follows.

By the dividing plane: $x = xm$ (xm is the x coordinate of the center of O_k), *Process x* divides O_k into two intermediate cubes O'_0, O'_1 and subdivides the polygons in Ξ_k into two lists: Ξ'_0 and Ξ'_1. The new centers of the intermediate cubes are obtained by the formulas, respectively:

$$\begin{aligned} xm'_0 &= xm - 1/2^{l+1},\ ym'_0 = ym,\ zm'_0 = zm, \\ xm'_1 &= xm + 1/2^{l+1},\ ym'_1 = ym,\ zm'_1 = zm \end{aligned} \tag{3.2}$$

where l is level number. Thus we have $O'_0 = \{0, 1, xm'_0, ym'_0, zm'_0, \Xi'_0\}$ and $O'_1 = \{1, 1, xm'_1, ym'_1, zm'_1, \Xi'_1\}$ where O'_0 is closer to the origin than O'_1. Note that 0 and 1 in the notations are binary digits.

Similarly, *process y* processes O'_0 using the dividing plane: $y = ym$, and produces two new intermediate-cubes O'_{00} and O'_{01}, where O'_{00} is nearer than O'_{01}. The formula for the center calculation is

$$\begin{aligned} xm'_0 &= xm',\ ym'_0 = ym' - 1/2^{l+1},\ zm'_0 = zm', \\ xm'_1 &= xm',\ ym'_1 = ym' + 1/2^{l+1},\ zm'_1 = zm', \end{aligned} \tag{3.3}$$

where (xm', ym', zm') is the center of O'_0. $y = ym$ is then used again to subdivide O'_1 and obtain O'_{10} and O'_{11}.

Finally, using the dividing plane $z = zm$, *process z* works on O'_{00}, O'_{01}, O'_{10} and O'_{11}, and outputs 8 intermediate-cubes O'_{000}, O'_{001}, ..., O'_{111} where all the centers are calculated based on:

$$xm'_0 = xm', ym'_0 = ym', zm'_0 = zm' - 1/2^{l+1},$$
$$xm'_1 = xm', ym'_1 = ym', zm'_1 = zm' + 1/2^{l+1}. \quad (3.4)$$

After the three processes are over, the last 8 intermediate cubes O'_b ($b_{(2)} = 000$, ..., 111) are *partial-solid* cubes O_{k_0}, O_{k_1}, ..., O_{k_7} where k_b (b=0,1,...,7) are octal numbers obtained by $k_{(8)} \times 10_{(8)} + b_{(8)}$. Note that k is the code of these new *partial-solid* cubes' parent.

After O_{k_0}, O_{k_1}, ..., O_{k_7} are obtained, O_k in ***AOL*** is replaced by them. So O_{k_0} becomes the first element in ***AOL***. The three processes are repeated with the current first cube in ***AOL***. If the current cube satisfies the given criteria, the cube is, therefore, a *final octant*. It is then removed from the ***AOL*** and appended to the ***FOL***. The subdivision algorithm terminates when ***AOL*** is empty. The detailed algorithm is shown as follows.

ALGORITHM 2 (3-D Subdivision Algorithm)

A PartialCube structure consists of six elements: $\{k, l, xm_k, ym_k, zm_k, \Xi_k\}$. Each of them is returned by the following functions: **CODE** returning k, **LEVEL** returning l, **CENTERX**, **CENTERY** and **CENTERZ** returning xm_k, ym_k and zm_k, respectively, and **POLYGON** returning Ξ_k. Other new functions not appearing in Algorithm 1 are **APPEND** that appends an element to a list, and **LAST** that returns the last element of a list.

PartialCube $O_0 \Leftarrow \{0, 0, 0.5, 0.5, 0.5, \Xi\}$; /* Ξ contains all polygons */
Stack of PartialCube ***AOL*** $\Leftarrow$ empty;
List of PartialCube ***FOL*** $\Leftarrow$ empty;

```
Subdivision3D()
{
PartialCube Ok,O'k,O'0,O'1;
PolygonList Ξk, Ξ'k, Ξ'0, Ξ'1;
double k, xm, ym, zm, k', xm', ym', zm';
int l;
double ratio;
List of PartialCube TOL ⇐ empty;

PUSH(O0 , AOL);

while( AOL is not empty ) {

    Ok⇐POP(AOL);
    k = CODE(Ok);
    xm = CENTERX(Ok); ym = CENTERY(Ok); zm = CENTERZ(Ok);
    l = LEVEL(Ok); Ξk= POLYGON(Ok);
    ratio = 1/2^(l+2);
    if (Ξk is not empty) {
        if(Ok meets the criteria) {
              APPEND{Ok, FOL };
        } else {

              process x:
              k0=2×k+0;
              k1=2×k+1;
              x0=xm-ratio; y0=ym; z0=zm;
              x1=xm+ratio; y1=ym; z1=zm;
              clip polygons in Ξk by the plane x = xm into two lists: Ξ'0 and Ξ'1;
              O'0⇐{k0,l+1,x0,y0,z0,Ξ'0};
              O'1⇐{k1,l+1,x1,y1,z1,Ξ'1};
              TOL ⇐{O'0, O'1};

              process y:
              for( i=0; i<2; i++ ){
                    O'k⇐CAR(TOL);
                    k'=CODE(O'k);
                    xm'=CENTERX(O'k); ym'=CENTERY(O'k); zm'=CENTERZ(O'k);
                    Ξk'=POLYGON(O'k);
                    k0= 2×k' +0;
                    k1= 2×k' +1;
                    x0 = xm'; y0 =ym'-ratio; z0 = zm';
                    x1 = xm'; y1 = ym'+ratio; z1 = zm';
                    clip polygons in Ξ'k by the plane y = ym into two lists: Ξ'0 and Ξ'1;
                    O0'⇐{k0,l+1,(x0, y0, z0),Ξ'0 };
                    O1'⇐{k1,l+1,(x1, y1, z0),Ξ'1 };
                    APPEND(O'0, TOL);
                    APPEND(O'1, TOL);
                    TOL ⇐ CDR(TOL);
              }

              process z:
              for( i=0; i<4; i++ ){
                    O'k⇐CAR(TOL);
                    k'=CODE(O'k);
```

```
                    xm'=CENTERX(O'k); ym'=CENTERY(O'k); zm'=CENTERZ(O'k);
                    Ξk'=POLYGON(O'k);
                    k0= 2×k' +0;
                    k1= 2×k' +1;
                    x0 = xm'; y0 =ym'; z0 = zm'-ratio;
                    x1 = xm'; y1 = ym'; z1 = zm'+ratio;
                    clip polygons in Ξ'k by the plane z = zm into two lists: Ξ'0 and Ξ'1;
                    O'0⇐{k0,l+1,x0,y0,z0,Ξ'0};
                    O'1⇐{k1,l+1,x1,y1,z1,Ξ'1};
                    APPEND(O'0, TOL);
                    APPEND(O'1, TOL);
                    TOL ⇐ CDR(TOL);
                }
                /* Push the newly derived partial-solid cube into AOL in right order */
                for( i=0; i<8; i++ )
                    PUSH( LAST(TOL), AOL );
            }
        }
    }
}
```

3.4 Specification of Criteria

The above algorithm indicates that an octant is appended to the ***FOL*** if it meets the criteria. Actually, how to specify criteria is an interesting subject. Whether a criterion is good or bad determines the accuracy of form-factors. In the following subsections, two simple criteria are described first, and then a more sophisticated one is introduced.

3.4.1 Simple Criteria

Ideally, the criterion should be specified by constraining the size of octants. When the dimension of an octant is less than a given size, the recursive subdivision on that octant stops. In this case, if the specified size is sufficiently small, all the polygons involved will eventually be subdivided into patches with the desired size because these patches are bounded in the final octants. This method, however, is too expensive. With a decrease in the specified size, the number of patches will increase too quickly to be handled by a conventional machine. If the size is large, it is difficult to obtain accurate radiosity. Another problem with this method is difficulty of determining depth priority of patches

within an octant if it contains more than one patch. The front-to-back algorithm described in Chapter 4 will fail to calculate the visibility of patches inside an octant. As a result, another hidden surface algorithm has to be employed.

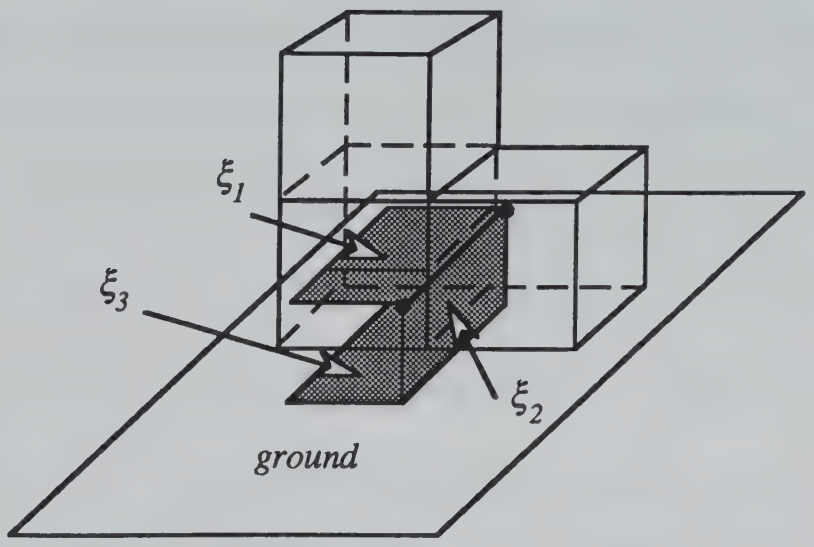


Figure 3.8. Failure of subdivision. Two polygons ξ_1 and ξ_2 cannot be separated by any cube until a very fine subdivision is obtained.

Another method for termination is to constrain both the size of octant and the number of polygons in it. When an octant is less than a specified size and contains only one patch, subdivision should terminate. This method results in another problem: the subdivision on some places, especially around edges shared by several polygons, might continue until any final octant contains a 3-D point. If an edge shared by polygons ξ_1 and ξ_2 almost coincides with a face of the cubic subdivision as shown in Figure 3.8, the subdivision never produces such a cube that contains only one of ξ_1 and ξ_2.

3.4.2 Criterion Vectors

Each of the above methods has its own advantages and disadvantages. A common drawback is exclusion of the adaptive subdivision feature. In spite of its imperfection, the following approach gives a better solution by including the adaptive subdivision.

A general criterion in this approach consists of a sequence of $\mathcal{M}$-dimensional vectors $C_i = (\mathcal{E}^i_1, \mathcal{E}^i_2, ..., \mathcal{E}^i_{\mathcal{M}})$ $(i = 1, ..., \mathcal{N})$ where each element $\mathcal{E}^i_j$ for $1 \leq j \leq \mathcal{M}$ characterizes a feature of the octant. The form of $\mathcal{E}^i_j$ is a logical expression including variables and constants. $\mathcal{E}^i_j$ is calculated by substituting dynamic values associated with the corresponding features for variables in $\mathcal{E}^i_j$. The value of a $\mathcal{E}^i_j$ is either true or false. An octant is said to be satisfactory with the criterion if only if the first *k-1* vectors satisfy

$$\mathcal{E}^i_1 \textbf{ and } \mathcal{E}^i_2 \textbf{ and } .... \textbf{ and } \mathcal{E}^i_{\mathcal{M}} = \textbf{FALSE} \qquad (i = 1, ..., k\text{-}1) \qquad (3.5)$$

and the *k*th vector satisfies

$$\mathcal{E}^k_1 \textbf{ and } \mathcal{E}^k_2 \textbf{ and } .... \textbf{ and } \mathcal{E}^k_{\mathcal{M}} = \textbf{TRUE} \qquad (\mathcal{N} \geq k \geq 1) \qquad (3.6)$$

where $\mathcal{M}$ is the number of features and $\mathcal{N}$ is the number of criteria. If an octant is satisfactory, the 3-D subdivision will not continue to subdivide the octant further but insert it into ***FOL*** (refer to **ALGORITHM 2**).

A perfect adaptive subdivision is impossible without an intelligent mechanism. On the other hand, achieving an optimal adaptive subdivision is possible if the criterion vectors are chosen carefully. Complex criterion vectors have impact on the performance of the 3-D subdivision since the criterion vectors have to be calculated every time a new cube or octant is obtained. The trade-off between optimization and feasibility need be considered in order to have a practical method.

Five features related to an octant are chosen to form the criterion vector: size of the octant, number of patches in the octant, parent relation of patches, joint relation between patches, and shadow test. It will become clear that the five-feature vector can yield a reasonable adaptive subdivision, especially around shadows and the object corners.

Let S be the size of an octant, N be the number of patches in an octant. $P(\mathcal{P})$ returns the parent polygon of the patch $\mathcal{P}$. $J(\mathcal{P}_1,\mathcal{P}_2)$ returns **TRUE** if $\mathcal{P}_1$ is joined with $\mathcal{P}_2$ by

a shared edge. $ST(\Xi_k)$ (*Shadow Test*) returns **TRUE** if Ξ_k contains at least a polygon that is partially lit where Ξ_k is the set of all patches in the current octant. A partially-lit polygon must contain a shadow boundary across its surface. Thus the shadow test can be formulated by

$$\exists \mathcal{P}_{lit}, \mathcal{P}\ (\boldsymbol{P}(\mathcal{P}_{lit}) == \boldsymbol{P}(\mathcal{P}) \textbf{ and } \sim \boldsymbol{V}(\mathcal{P}_{lit}, \mathcal{P})) \quad (\ \mathcal{P}_{lit} \in \Xi_k\text{-}lit,\ \mathcal{P} \in \Xi_k\ )$$

where $V(\mathcal{P}_1, \mathcal{P}_2)$ returns **TRUE** if $\mathcal{P}_1$ and $\mathcal{P}_2$ have exactly the same vertices, i.e., they are the same patch, and Ξ_k-*lit* is a set of lit patches similar to Ξ_k but derived from Ξ-*lit*. As mentioned in Section 3.2.2, for each vertex of each polygonal light, the preprocessing produces a corresponding list of lit subpolygons. ST is, therefore, calculated against all the Ξ_k-*lit*'s.

Note that if $N = 0$, the octant is simply discarded before it is tested against the criterion. With the above variables and functions, the criterion can be determined by the following sequence of criterion vectors.

The first vector is read as follows:

$$C_1 = (S \leq \textbf{SIZE1},\ N == 1,\ \textbf{TRUE},\ \textbf{TRUE},\ \sim ST(\Xi_k)),$$

where **SIZE1** is a constant that can be defined on the fly based on the scene being modeled, and **TRUE** indicates that the item is not used for this vector. This vector means that an octant O_k is satisfactory if the size of the octant is not greater than the value **SIZE1**, and the octant contains only one patch that must be not partially-lit. This vector can be used to obtain a regular subdivision in the interior of large polygon surfaces.

The second vector looks like

$$C_2 = (S \leq \textbf{SIZE2},\ N > 1,\ \textbf{TRUE},\ \forall \mathcal{P}_1 \mathcal{P}_2 \sim J(\mathcal{P}_1, \mathcal{P}_2),\ \sim ST(\Xi_k)) \qquad (\ \mathcal{P}_1,\ \mathcal{P}_2 \in \Xi_k\ )$$

where **SIZE2** is a pre-defined constant not greater than **SIZE1**. C_2 implies that an octant O_k is satisfactory if C_1 is not satisfied, the octant size is not greater than **SIZE2** ($\leq$**SIZE1**), the

octant contains more than one patch, any pair of patches in the octant are not joined by a shared edge, and there is no partially-lit patch in the octant. This vector is dedicated to those large surfaces that occur close together. Crowded polygons deserve to get a subdivision finer than a stand-alone polygon as occurred in C_1. The finer subdivision is reflected by the number **SIZE2** that is smaller than **SIZE1**.

The third vector is formed by

$$C_3 = (S \leq \textbf{SIZE3}, N>1, \forall \mathcal{P}_1\mathcal{P}_2(P(\mathcal{P}_1)\neq P(\mathcal{P}_2)), \exists \mathcal{P}_1\mathcal{P}_2 J(\mathcal{P}_1,\mathcal{P}_2), \sim ST(\Xi_k))\ (\ \mathcal{P}_1, \mathcal{P}_2 \in \Xi_k\ )$$

where **SIZE3** $\leq$ **SIZE2**. This vector says that an octant O_k is satisfactory if C_1 and C_2 are not satisfied, the octant size is not greater than **SIZE3**, the octant contains more than one patch, all patches have no common parent polygon from which they originate, there exist at least one pair of patches that share an edge, and there is no partially-lit patch in the octant. This vector can be used to perform an extensive subdivision around edges or corners of objects if **SIZE3** is relatively small. This operation, therefore, achieves an adaptive subdivision around edges and corners that often have high intensity gradients.

The last vector takes the following form:

$$C_4 = (S \leq \textbf{SIZE4}, N>1, \textbf{TRUE}, \textbf{TRUE}, ST(\Xi_k))$$

where **SIZE4** $\leq$ **SIZE3**. This vector characterizes such an octant that contains a partially-lit patch. As a result, C_4 makes it possible to achieve extensive subdivision around shadow boundaries if **SIZE4** is chosen to be sufficiently small.

The above four tests can fail even when the octant size is very small. In order to terminate the 3-D subdivision at a reasonable level, a small number **SIZE5** ($\leq$**SIZE4**) is used to test the octant size after C_4. If the octant size is smaller than **SIZE5**, the subdivision stops unconditionally.

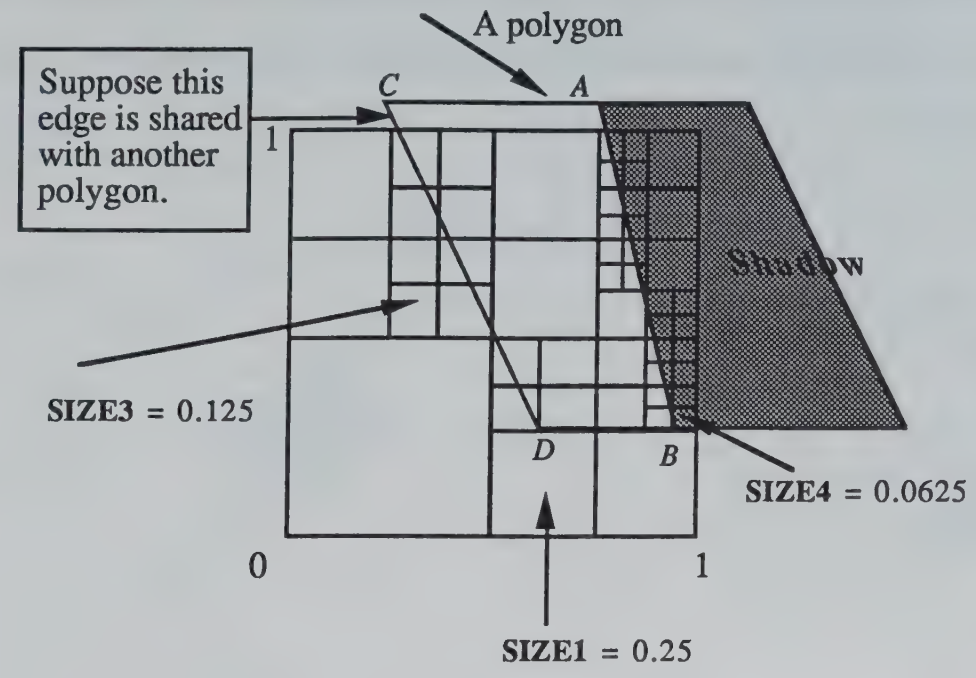


Figure 3.9. Extensive subdivision is performed around the shadow edge *AB* and the polygon edge *CD*. (The picture only shows a cross section of the cubes of the octree.)

Figure 3.9 shows an example of the extensive subdivision around the shadow edge *AB* and the polygon edge *CD* where **SIZE1**=0.25, **SIZE2**=0.2, **SIZE3**=0.125 and **SIZE4**=0.0625.

To sum up, the criterion with these four vectors achieves four types of adaptive subdivisions in sequence: (1) regular subdivision on a stand-alone large patch, (2) slightly finer subdivision on crowded large patches, (3) extensive subdivision around object's edges, and (4) more extensive subdivision around shadow boundaries.

3.4.3 Limitations of Criterion Vector

Although the above method produces a good adaptive subdivision, it still has the following limitations that prevents the 3-D subdivision from achieving truly adaptive subdivision.

In the situation (a) shown in Figure 3.10, the large polygon $\mathcal{P}_1$ behind the shadowed one $\mathcal{P}_2$ obtains an unexpected extensive subdivision. This is because $\mathcal{P}_1$ is so

close to $\mathcal{P}_2$ that the cubes derived through the 3-D subdivision is unable to separate them; therefore, $\mathcal{P}_1$ obtains the same subdivision as that of the shadowed polygon $\mathcal{P}_2$.

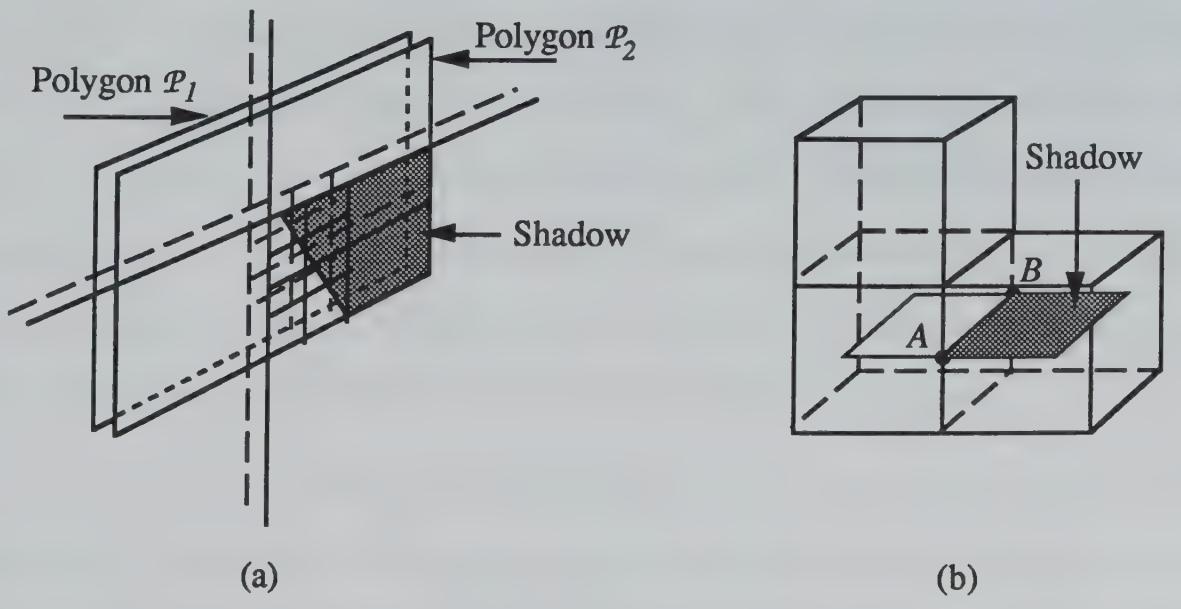


Figure 3.10. Two cases of incorrect subdivisions. (a) Unexpected extensive subdivision on the flat polygon $\mathcal{P}_1$. (b) A cube cuts exactly through the shadow edge *AB*.

Another problem is that a cube just cuts exactly through the shadow edge *AB* as shown in Figure 3.10 (b). In this case, the test ***ST*** returns **FALSE** because the patch on the left side of *AB* is completely lit. In this case, $\mathcal{P}$ may end up with a regular subdivision. Generally speaking, this problem is not very serious since the situation where a face of cube cuts exactly through an shadow edge seldom happens.

Calculating the vectors is another concern. Expressions with operators $\exists$ or $\forall$ are very expensive computationally because they require searching patches in the octant. Fortunately, checking the octant size is always done before these complex expressions are calculated. The testing of the octant size fails in most of cases. Thus calculating complex expressions is performed only when the size of octant is sufficiently small. A small octant

normally contains a few patches that do not add much overhead in calculating these expressions. As a result, this problem is not serious.

The preprocessing produces lists of lit polygons. Only those polygons visible to the primary light sources are identified in the preprocessing. Secondary light sources such as shining surfaces occur at many places in reality. The above adaptive subdivision using the vector C_4 fails to give an extensive subdivision around the shadows cast from these secondary lights. Calculating these shadows is almost impossible before intensity for each patch is actually obtained. Chapter 8 briefly describes an approach applying the 3-D subdivision to the progressive refinement which may solve this problem.

The vector approach for adaptive subdivision can be improved if the 3-D subdivision is an interactive system allowing vector expressions to be specified on the fly. Interaction with the designer can give more accurate adaptive meshing of patches. Furthermore, optimizing criterion vectors can also improve the adaptive subdivision. How to optimize vectors or find a proper vector is a difficult and interesting study. It is worthy of further investigation. These improvements are beyond the scope of this thesis and we leave them for any interested professional.

Due to the lack of time, the shadow clipper described in Section 3.2.2 was not implemented. C_1, C_2 and C_3 were, therefore, implemented with the shadow test removed, and C_4 is replaced with one similar to C_3. To achieve an extensive subdivision around shadows, we first manually split each possibly-shadowed polygon along the boundary of the shadow into several subpolygons. Thus these split subpolygons are joined but have the same parent. The modified C_4 becomes

$$C_4 = \exists \mathcal{P}_1\mathcal{P}_2(\boldsymbol{S}\leq \mathbf{SIZE4}, \boldsymbol{N}>1, \boldsymbol{P}(\mathcal{P}_1)==\boldsymbol{P}(\mathcal{P}_2), \boldsymbol{J}(\mathcal{P}_1,\mathcal{P}_2)) \quad (\mathcal{P}_1, \mathcal{P}_2 \in \Xi_k).$$

To end this chapter, a complete result of the 3-D subdivision for a simple scene is shown in Figure 3.11. For simplicity, only a point light instead of area lights is used in this example.

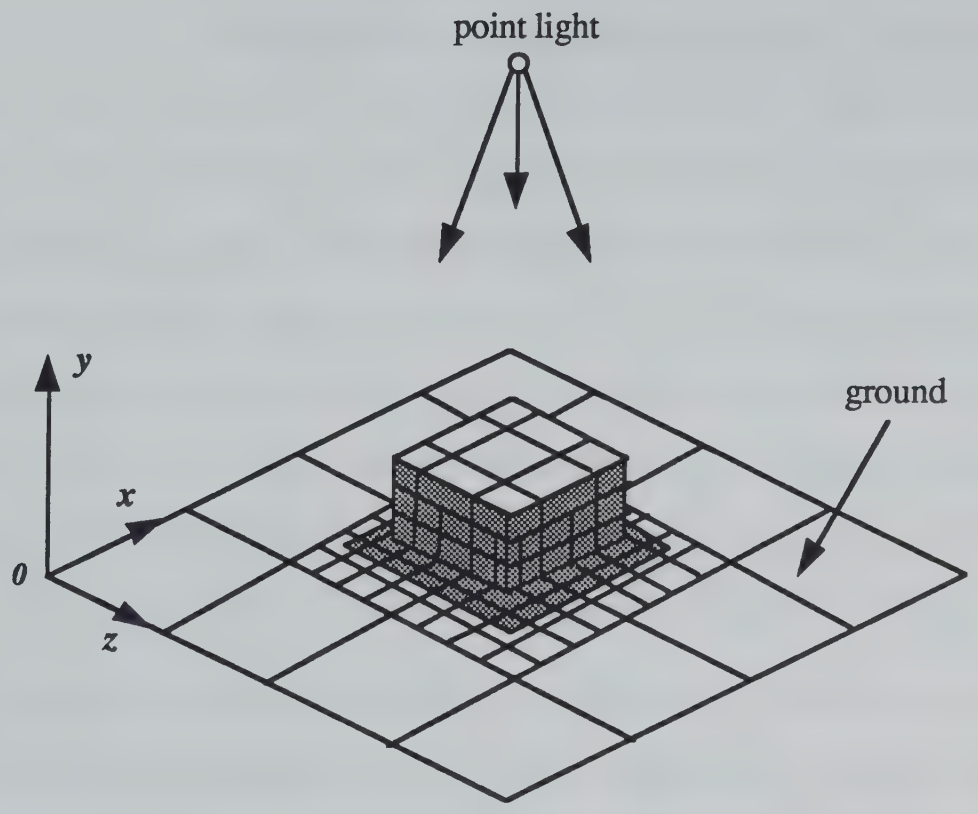


Figure 3.11. An example of polygon subdivision in a simple scene using the 3-D subdivision.

Chapter 4

Visibility in Form-factor Computation

A form-factor from one patch to another describes the fraction of energy leaving the first one and reaching the second. The energy fraction is determined by a purely geometric relation between the pair of patches plus a visibility factor. Determining visibility of one patch to another is not an easy job. It requires sophisticated hidden surface removal algorithms. Since the number of form-factors required for radiosity solutions is N^2, calculating all the form-factors involves a lot of visibility computation. Therefore, fast visibility computation is critical to increase the efficiency of radiosity methods.

To accelerate visibility determination, the **FBRM** first establishes an octant priority based on the linear octree constructed in the 3-D subdivision, and then implements a modified hemi-cube method with a front-to-back hidden surface algorithm. This chapter will discuss these aspects in detail.

4.1 Octant Priority

The 3-D subdivision produces a linear octree with a list of patches: $P = \{\mathcal{P}_1, \mathcal{P}_2, ..., \mathcal{P}_N\}$, each of which is contained in one of the octree's leaves. To perform visibility computation, these leaves, or octants, need be sorted on depth with respect to a given viewpoint. The ordering of the octants is called *octant priority*. Since the octant priority is similar to Schumacker's cluster priority, the Schumacker visibility algorithm [ScBr69, SuSp74] is described first.

4.1.1 Cluster Priority in Schumacker Algorithm

Hidden surface removal is one of the most difficult problems in image synthesis. Calculating depth priority is the core of hidden surface removal. In 1969, Schumacker proposed a fast algorithm for determining hidden surfaces as a viewer position changes constantly. In the Schumacker algorithm [ScBr69], all polygons in a scene are first collected into clusters of polygons, which are linearly separable; that is, any two clusters can be separated by a dividing plane. These dividing planes can be represented by a binary tree in which the internal nodes are the dividing planes and the leaves are the regions partitioned by the dividing planes. In the tree, each node has two branches. The left branch, notated by "+", stands for the half space on the positive side of the plane associated with the node, while the right branch, notated by "–", stands for the other half space on the negative side of the plane. Each region at the leaf is associated with a pre-calculated priority order in which clusters obscure each other when the viewpoint is located in that region. An example of the binary tree is given in Figure 4.1 (b).

Given an arbitrary view location, the cluster priority with respect to the viewpoint can be found by traversing the binary tree as follows. At each node, the branch that is visited depends on the order in which clusters are displayed. If a front-to-back paint algorithm[1] is used, the branch associated with the side of the dividing plane on which the viewer lies is visited. The traversal is carried on until a leaf is reached. The pre-calculated cluster priority associated with the leaf is then the result. In the example shown in Figure 4.1 where the viewer is located in area *D*, at the first level, the branch marked by "–" is visited; then "–" is visited at the second level. Thus the cluster priority is 2, 1, 3.

---

[1]Throughout this thesis, the front-to-back strategy is always assumed.

4.1.2 Formation of Octant Priority

The 3-D subdivision algorithm, as indicated in the preceding chapter, uses a sequence of virtual planes to continuously partition the space into octants and clip the polygons into patches. Each octant contains a group of patches, and each pair of octants are separated by such a virtual plane. Therefore, an octant can be regarded as a cluster of patches, and the virtual planes as the dividing planes for separating clusters as was done in the Schumacker algorithm. By applying an algorithm similar to the Schumacker one, an octant priority can be formed.

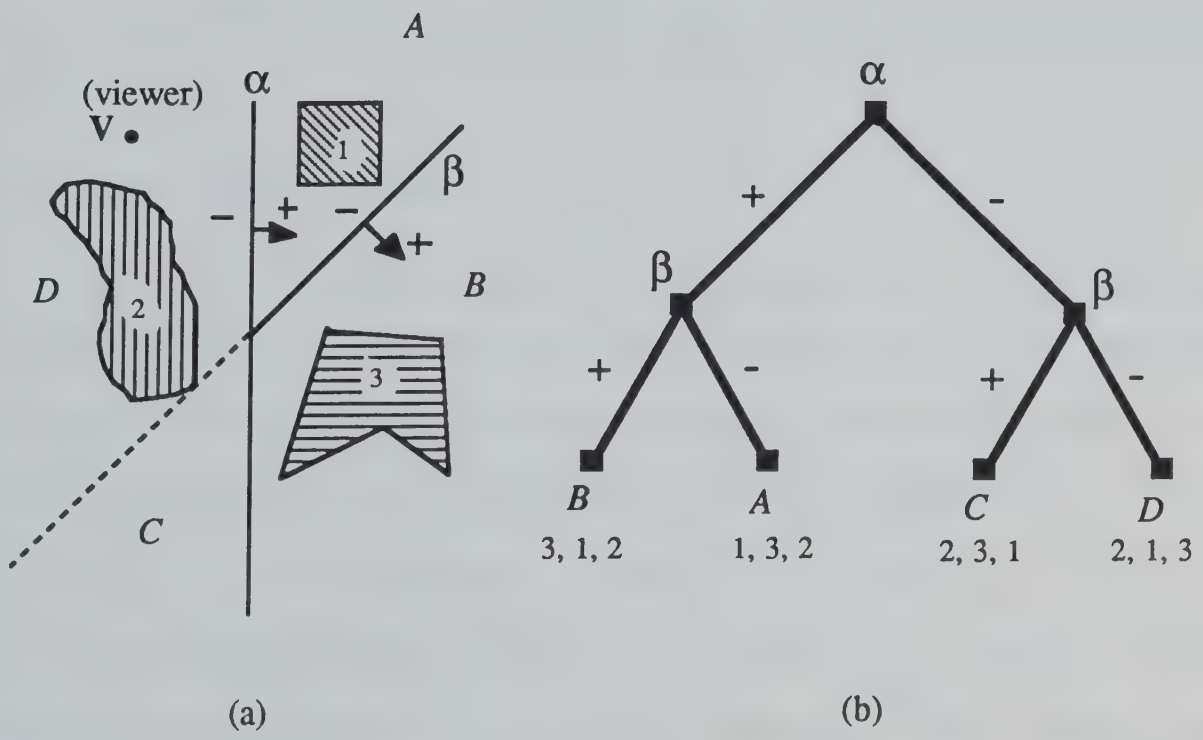


Figure 4.1. Schumacker's cluster priority. (a) Two dividing planes α and β separate three clusters into four regions. (b) The corresponding binary tree.

Actually, octant priority is a special case of cluster priority. An octant has regular-shaped boundaries, while a cluster has irregular boundaries. Because of these special characteristics, modifications to the original Schumacker algorithm have been made. The

binary tree is replaced by the octree produced in the 3-D subdivision, in which each node has at most 8 children instead of 2.

To determine priority of octants is in fact to sort the octants by their distances from the given viewpoint. If the Schumacker's strategy is applied, sorting octants needs to traverse the octree from the top to the bottom by visiting the child of each node that is closest to the viewpoint. Based on the characteristics of the linear octree, two modifications are made: (1) traversing the octree is performed from the bottom to the top; (2) no pre-calculated octant priority need be associated with each leaf of the octree so that a great amount of memory is saved. With the modifications, the operation at each node is to sort the node's children into a so-called *basic octant priority* instead of finding the closest child.

4.1.2.1 Basic Octant Priority

Let $O_k = \{k, l, xm, ym, zm, P_k\}$[1] be a partial-solid cube. Considering Figure 4.2, when the cube O_k is subdivided into 8 smaller octants $O_{k_0}, O_{k_1}, ..., O_{k_7}$ by 3 planes $x=a$, $y=b$, $z=c$ (a, b, c are constants), the space is also divided into 8 sub-spaces, S_{k_o}, S_{k_1} ..., S_{k_7}. According to the numbering scheme introduced in Section 3.3.1, k_i is equal to $k*8+i$ ($i=0,...,7$). If the octal system is used and $k = (b_1b_2...b_r)_8$, then $k_i = (b_1b_2...b_ri)_8$.

Suppose that the viewpoint is in S_{k_3}. If the plane $x = a$ is considered, ignoring others, it is clear that any patch lying on the side of $x = a$ further from the origin cannot obscure the patches on the side closer to the origin; therefore, $O_{k_0}, O_{k_1}, O_{k_2}, O_{k_3}$ should have higher priority than $O_{k_4}, O_{k_5}, O_{k_6}, O_{k_7}$. Using the same principle, when the planes $y=b$ and $z=c$ are considered in turn, each of the above two groups is sorted again,

---

[1] In Chapter 3, $O_k = \{k, l, xm, ym, zm, \Xi_k\}$ represents a partial-solid cube or an octant where the notation Ξ_k is used since *polygons* were the subjects. From now on, patches will be the subject; therefore the notation P_k is used.

respectively, and finally the priority of the *8* octants can be determined as follows: O_{k_3}, O_{k_2}, O_{k_1}, O_{k_0}, O_{k_7}, O_{k_6}, O_{k_5}, O_{k_4}.

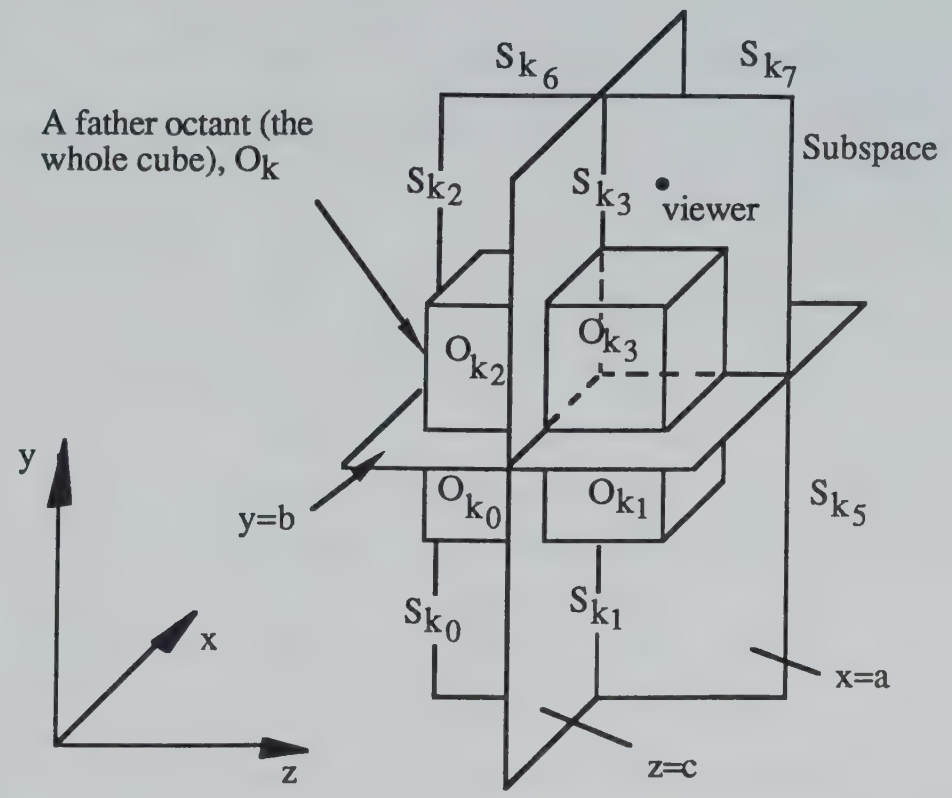


Figure 4.2. Space subdivision.The space is divided into 8 sub-spaces by the three dividing planes. The octant priority can be determined according to the sub-space in which the viewer is located.

Note that the viewing position was not particularly specified in S_{k_3} when the priority of O_{k_0}, ..., O_{k_7} was derived. Therefore, the above ordering, O_{k_3}, O_{k_2}, O_{k_1}, O_{k_0}, O_{k_7}, O_{k_6}, O_{k_5}, O_{k_4}, remains unchanged for any viewing position within S_{k_3}. Thus, there are only *8* orderings, each corresponding to a sub-space in which the viewer lies. These 8 orderings can be pre-computed and stored in a table. This table is called the *basic octant priority table* (***BOPT***). The complete table is listed in Table 1.

During the bottom-to-top traversal of the octree, the following simple calculation is performed at each node to determine the relation of the viewer with sub-spaces.

$$b = (view.x >= xm)*4 + (view.y >= ym)*2 + (view.z >= zm) \qquad (4.1)$$

where (xm, ym, zm) is the center of the partial-solid cube associated with the node. S_{k_b} is the sub-space in which the viewer lies. Using b to index ***BOPT***, the order of the 8 children of the node can be found.

TABLE 1 (Basic Octant Priority)

```
BOPT[8][8] = {
                {0, 1, 2, 3, 4, 5, 6, 7},
               {1, 0, 3, 2, 5, 4, 7, 6},
               {2, 3, 0, 1, 6, 7, 4, 5},
               {3, 2, 1, 0, 7, 6, 5, 4},
               {4, 5, 6, 7, 0, 1, 2, 3},
               {5, 4, 7, 6, 1, 0, 3, 2},
               {6, 7, 4, 5, 2, 3, 0, 1},
               {7, 6, 5, 4, 3, 2, 1, 0}
              }
```

4.1.2.2 Bottom-to-Top Traversal of the Linear Octree

Let $\boldsymbol{FOL} = \{O_{k_1}, O_{k_2}, ..., O_{k_M}\}$ represent the linear octree where $O_{k_i} = \{k_i, l_i, xm_i, ym_i, zm_i, P_{k_i}\}$. As indicated in Section 3.3.3, the 3-D subdivision guarantees that $O_{k_1}, O_{k_2}, ..., O_{k_M}$ are arranged in a *natural* order; that is, the "*brothers*" having the same parent in the octree appear next to each other in the ***FOL***. Let $B = [O_{k_i}, O_{k_{i+1}}, ..., O_{k_{i+H}}]$ $(1 \le i \le M, 0 \le H \le 7)$ represent a group of *brothers*. Thus the ***FOL*** can be rewritten as follows,

$$\boldsymbol{FOL} = \{B_1, B_2, ..., B_G\},$$

where $0 \le G \le M$.

With these notations, it is not difficult to prove the following properties of *brother octants* in the ***FOL***.

Property 1 If O_{k_i} and O_{k_j} are brothers in B_g, then $l_i = l_j$. Thus we define $L_g = l_i$ to be the level of B_g.

Property 2 If O_{k_i} and O_{k_j} are brothers, then $0 \le k_i - k_j \le 7$. If k_i and k_j are represented by octals, k_i differs from k_j only at their last octal digit. In other words, if $k_i = (b_1 b_2 .. b_{r-1} b_r)_8$, thus $k_j = (b_1 b_2 .. b_{r-1} x)_8$ where $x \neq b_r$ and $0 \le x \le 7$. The octal number $(b_1 b_2 .. b_{r-1})_8$ is the code of their parent cube, which is returned by the function **PCODE** appearing later.

Based on these properties, the following algorithm is designed for sorting the octants in ***FOL*** for any given viewpoint. It is called the *octant sort algorithm*. The main operations in the algorithm are **Parent** and **BasicPriority**. The **Parent** returns the parent cube of a group of brothers. A parent cube is represented by a vector $PC_k = (k, l, xm, ym, zm, Os_k)$ where k is the code for the cube, l is the level of the cube, (xm, ym, zm) is the center of the cube and Os_k is a list of all the octants derived from the cube. The **BasicPriority** sorts a group of brothers using the *basic octant priority table* (***BOPT***).

The algorithm first maps each octant O_{k_i} in ***FOL*** into a form of the parent cube $PC_{k_i} = (k_i, l_i, xm_i, ym_i, zm_i, \{O_{k_i}\})$. In this case, each group of brothers, B_g (g=1,...,G), is a collection of PC's. Suppose that the maximum level of the octree is D, that is, $D = \mathbf{max}(L_1, L_2, ..., L_G)$. The algorithm repeats the following steps for D times. (The variable d is initialized to D.)

STEP1 Find all groups of brothers, B_g in ***FOL***, such that L_g is equal to d.

For each of such groups: $B_g = [PC_{k_i}, ..., PC_{k_{i+h}}]$ $(0 \le h \le 7)$,

STEP2 Apply **BasicPriority** to sort B_g. Suppose $[PC_{k_i}, ..., PC_{k_{i+h}}]$ is already in order.

STEP3 Apply **Parent** to return B_g's parent: $PC_{k'} = (k', d\text{-}1, xm', ym', zm', Os_{k'})$ where k' is the code of the parent cube, (xm', ym', zm') is the center of the cube. $Os_{k'}$ is the catenation of Os_{k_i}, ..., $Os_{k_{i+h}}$

STEP5 Replace B_g in ***FOL*** by $PC_{k'}$ and merge B_{g-hl}, ..., B_{g-1}, $PC_{k'}$, B_{g+1}, ..., B_{g+hr} if they are brothers, where $0 \le h_l, h_r \le 7$ and $h_l+h_r \le 7$.

STEP6 $d = d - 1$, goto step 1.

The center of the parent cube, (xm', ym', zm'), can be obtained from the center of any of its children: $O_k = (k, l, xm, ym, zm, P_k)$,

$$xm' = xm + 1/2^{l+1} \times (1\text{-}2\times(1\&(k>>2))),$$
$$ym' = ym + 1/2^{l+1} \times (1\text{-}2\times(1\&(k>>1))),$$
$$zm' = zm + 1/2^{l+1} \times (1\text{-}2\times(1\&k)). \qquad (4.2)$$

ALGORITHM 3 (Octant Sort)

A ParentCube structure consists of six elements: $\{k, l, xm_k, ym_k, zm_k, Os_k\}$. The function **OCTANT** returns Os_k.

List of PartialCube ***FOL*** $\Leftarrow \{O_{k_1}, O_{k_2}, ..., O_{k_M}\}$;

```
OctantSort()
{
ParentCube PC, B[8];
List of ParentCube TPCL ⇐ empty;

/* map octant into ParentCube */
for( i=0; i<NUMBER(FOL); i++ ) {
    PC ⇐ {k_i, l_i, xm_i, ym_i, zm_i, {O_ki}};
    FOL ⇐ CDR(FOL);
    APPEND( PC, FOL );
}

for( d=D; d>0; d-- ) {
    Num = NUMBER(FOL);
    for( i=0; i<Num; i++ ){
        if( d == LEVEL(CAR(FOL)) ) {
            j=0;
            do(
                B[j] ⇐ CAR(FOL);
                FOL ⇐ CDR(FOL);
                j++; i++;
```

```
            while( d == LEVEL(CAR(FOL));
            BasicPriority(B, j, d);
            PC ⇐ Parent(B, j, d);
            APPEND(PC, TPCL);
        } else {
            APPEND(CAR(FOL), TPCL);
            FOL ⇐ CDR(FOL);
        }
    }
    FOL ⇐ TPCL;
}
}

BasicPriority(B, h, d)
ParentCube B[];
int h, d;
{
int i, j, l, k;
double xm, ym, zm;
ParentCube TB[8];

    /* obtain the center of parent cube */
    k = CODE(B[0]);
    xm = CENTERX(B[0]) + 1/2^(d+1) × (1-2×(0x1&(k>>2)));
    ym = CENTERY(B[0]) + 1/2^(d+1) × (1-2×(0x1&(k>>1)));
    zm = CENTERZ(B[0]) + 1/2^(d+1) × (1-2×(0x1&k));

    /* calculate the index to BOPT. */
    b = (view.x ≥ xm)×4 + (view.y ≥ ym)×2 + (view.z ≥ zm);

    /* sort B[] and place the sort result in TB[] */
    l = 0;
    for(i=0; i<8; i++) {
        for(j=0; j<h, j++) {
            if( BOPT[b][i] == (0x7&CODE(B[j])) )
                TB[l++] ⇐ B[j];
        }
    }

    /* transfer the sort result back to B[] */
    for(j=0; j<h, j++)
        B[j] = TB[j];
}

ParentCube Parent(B, h, d)
ParentCube B[];
int h, d;
{
List of PartialCube Os, Os';
ParentCube PC;
int k, k', l';
double xm', ym', zm';

    k = CODE(B[0]);
    k' = PCODE(B[0]);
    l' = d-1;
    xm' = CENTERX(B[0]) + 1/2^(d+1) × (1-2×(0x1&(k>>2)));
```

```
        ym' = CENTERY(B[0]) + 1/2^(d+1) × (1-2×(0x1&(k>>1)));
        zm' = CENTERZ(B[0]) + 1/2^(d+1) × (1-2×(0x1&k));

        /* merge all the octants ascended from the parent cube */
        for(i=0; i<h; i++) {
            Os ⇐ OCTANT(B[i]);
            for(j=0; j<NUMBER(Os); j++) {
                APPEND(CAR(Os), Os');
            }
        }
        PC ⇐ {k', l', xm', ym', zm', Os'};
        return PC;
    }

    PCODE(B)
    ParentCube B;
    {
    int k;

        k = CODE(B);
        return (k>>3);

    }
```

To demonstrate the process of the octant sort, let us revisit the octree shown in Figures 3.6 and 3.7. Suppose that the viewpoint is located inside the cube β in Figure 3.6 (b). The bottom-to-top traversal of the octree is shown by a sequence of trees in Figure 4.3. In the first tree, the leaf nodes are three groups of brothers, each containing octants sorted by **BasicPriority**. In the second tree, a new group of brothers are formed by the bottom-to-top traversal, and these brothers are sorted by indexing ***BOPT*** with b=1. The last tree shows the final result of the sort. The order in that the numbers appear is the priority of these octants represented by the numbers.

4.2 Basis of Hemi-Cube Method

After octant priority is obtained, a hidden surface removal algorithm is used to perform the form-factor computation. The hemi-cube method introduced in [CoGr85] is widely accepted because of its feasibility. As seen in Chapter 2, many radiosity techniques have used it as the basic method for calculating form-factors. The new method **FBRM** is also based on the hemi-cube. Therefore, it is worthwhile to examine it in detail.

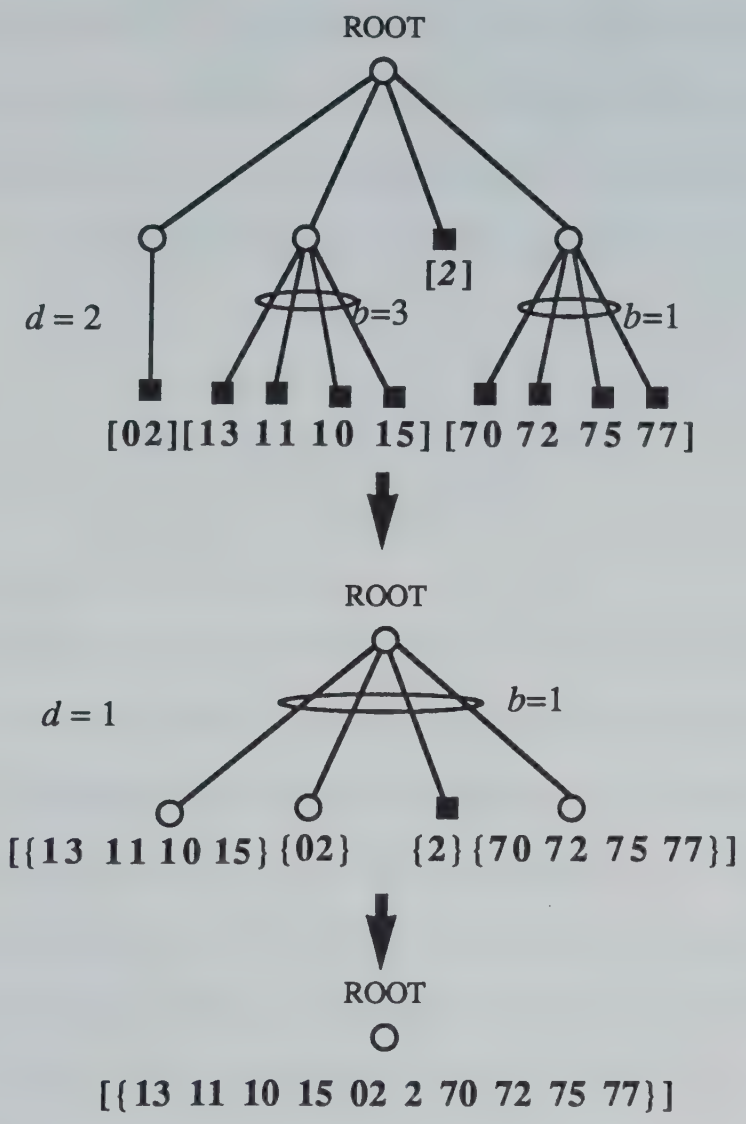


Figure 4.3. The process of octant sort for the octree shown in Figure 3.7.

4.2.1 Hemisphere

The hemi-cube method differs from the other methods mainly in calculating form-factors. More precisely, the difference lies in the numerical approximation to (2.13). The following illustration is necessary to understand the hemi-cube and **FBRM** as well. (2.13) is repeated here.

$$F_{ij} = \int_{A_i} \int_{A_j} \frac{\cos \phi_i \cos \phi_j}{\pi A_i r^2} dA_j dA_i \,. \qquad (2.13)$$

Let the inner integral in (2.13) be represented by a delta-to-finite form-factor F_{dij}. Thus, $F_{ij} = 1/A_i \int_{A_i} F_{dij} dA_i$. It is evident that F_{ij} is approximately equal to the average value of the delta-to-finite form-factors in the area A_i. If the finite area is sufficiently small, the delta-to-finite form-factor at the center point of A_i will give a good approximation to the average value. Therefore, the double integral is approximated by the inner integral at the center of A_i:

$$F_{ij} \approx F_{dij} = \frac{1}{\pi} \int_{A_j} \frac{\cos \phi_i \cos \phi_j}{r^2} dA_j \,, \tag{4.3}$$

where d_i is the index of the center on A_i.

The geometric analog for the inner integral, that was proposed by Nusselt in 1978 [SiHo78], has existed for years. Figure 4.4 intuitively demonstrates the process of the Nusselt analog. By analyzing results, Nusselt stated that a centered delta-to-finite form-factor is equal to the fraction of the base of a hemisphere covered by projecting the finite area A_j onto the hemisphere and then orthographically down onto the base.

The correctness of Nusselt's statement directly follows from the following brief considerations. The projection onto the hemisphere has an effect equivalent to the $1/r^2$ term times the cosine of the angle between the normal of the projecting patch and the r vector. The orthographic projection onto the base is equivalent to the product of the first projection and the cosine of the angle between the normal of the base and the r vector. As a result, dividing the area of the second projection by the base area with unit radius which is equal to π yields the value for the form-factor F_{ij}.

Calculating the area of the second projection is not an efficient method to obtain the form-factor. It is only an analytic method, but it is useful for deriving the following hemi-cube approach.

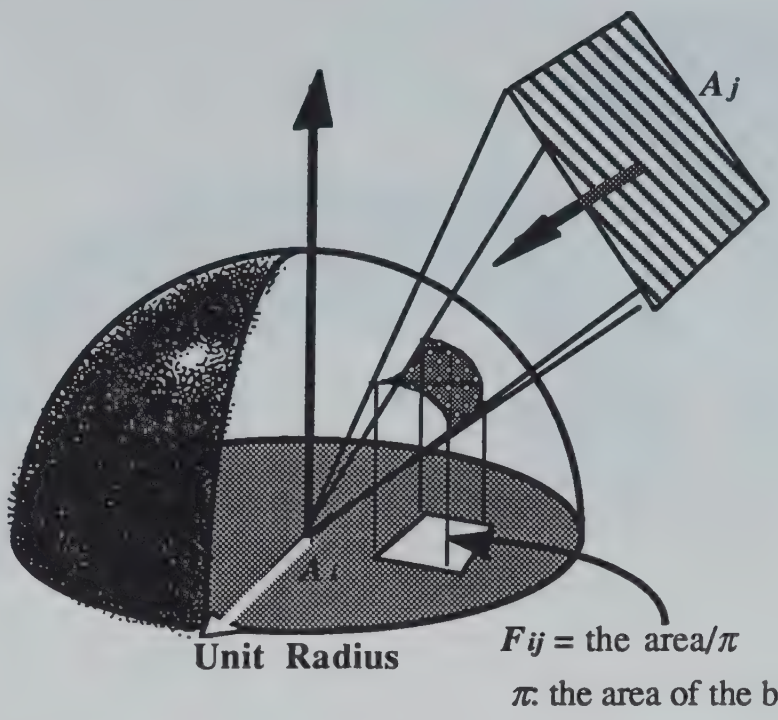


Figure 4.4. Hemisphere approximation to the form-factor.

4.2.2 Hemi-Cube

Consider a unit hemi-cube mounted on the patch A_i as shown in Figure 4.5. Thus, the hemisphere remains inside the hemi-cube. For descriptive convenience, the face of the hemi-cube perpendicular to the normal is called the top face and the others are called side faces. Let A'_j represent the projection of the finite area A_j onto the hemi-cube. (Note that the entire A'_j does not necessarily cover only one face of the hemi-cube. A'_j should be equal to the sum of projections on all the five faces of the hemi-cube, as shown in Figure 4.5.) It is evident that A'_j and A_j have the same projection onto the hemisphere. Thus the orthographic projection onto the circle base are identical for both A'_j and A_j. This implies that the form-factor from A_i to A'_j is the same as that from A_i to A_j. Therefore, the original form-factor computation for A_i to A_j is converted to calculate the form-factor between A_i and A'_j.

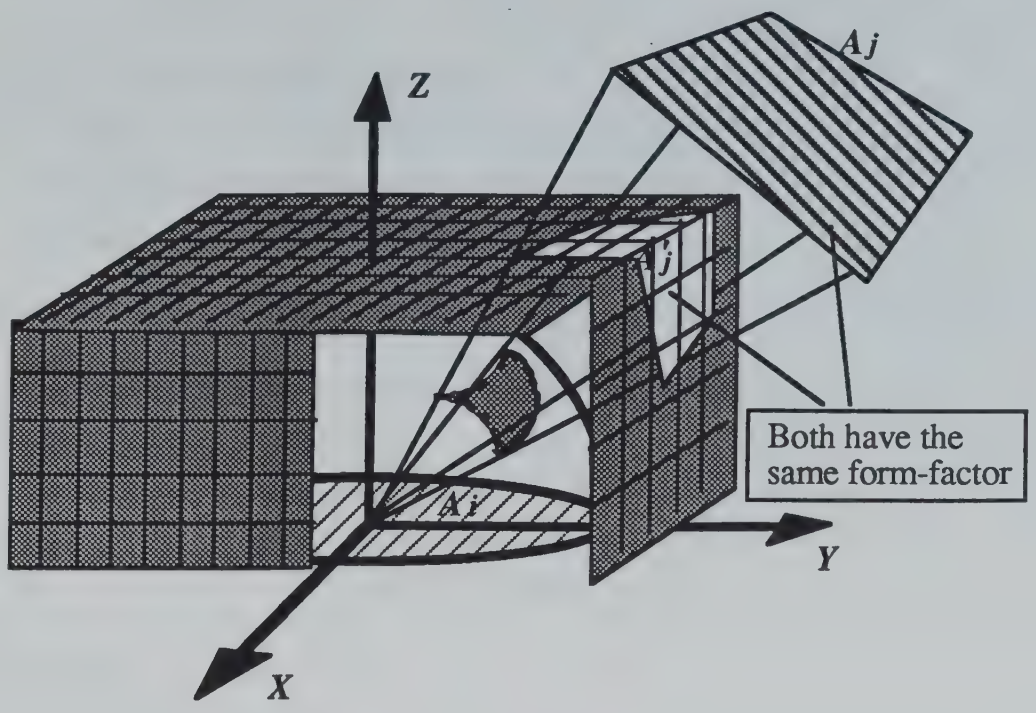


Figure 4.5. Hemi-cube approximation to the form-factor.

To calculate the A_i to A'_j form factor, the hemi-cube is discretized into a 2-D array of squares called *pixels* on each face. If $\Delta a_j(x,y,z)$ represents the area of the pixel centered at (x,y,z), the projection on the hemi-cube is

$$A_j^{'} = \sum_{q=1}^{n} \Delta a_j(x_q,y_q,z_q) \tag{4.4}$$

where n is the number of pixels covered by the projection. The form factor from A_i to A_j' is then related by

$$\begin{aligned} F_{ij}^{'} &= \frac{1}{\pi}\int_{A_j^{'}} \frac{\cos\phi_i \cos\phi_j^{'}}{r^2}\, dA_j^{'} \\ &\doteq \sum_{q=1}^{n} \Delta F_q \end{aligned} \tag{4.5}$$

where

$$\Delta F_q = \frac{\cos\phi_{i_q}\cos\phi_{j_q}}{\pi r_q^2}\,\Delta a_j(x_q,y_q,z_q). \tag{4.6}$$

and

(x_q,y_q,z_q) — the center of the qth pixel;

r_q — the distance between the center of A_i and (x_q,y_q,z_q);

ϕ_{iq} — the angle between the surface normal to A_i and r_q;

ϕ_{jq} — the angle between the surface normal to A'_j and r_q.

ΔF_q can be determined directly by calculating the above expression. The calculation of (4.6) for the top of the hemi-cube is different from the side. For the top,

$$r_q = \sqrt{x_q^2 + y_q^2 + z_q^2} = \sqrt{x_q^2 + y_q^2 + 1},$$

$$\cos\phi_{i_q} = \cos\phi_{j_q} = \frac{1}{r_q}.$$

Thus, from (4.6),

$$\Delta F_q = \frac{\Delta a_j(x_q,y_q,z_q)}{\pi(x_q^2 + y_q^2 + 1)^2}. \tag{4.7}$$

Similarly, for a side face, e.g. $y = \pm 1$,

$$r_q = \sqrt{x_q^2 + y_q^2 + z_q^2} = \sqrt{x_q^2 + 1 + z_q^2},$$

$$\cos\phi_{i_q} = \frac{z_q}{r_q}, \qquad \cos\phi_{j_q} = \frac{1}{r_q},$$

$$\Delta F_q = \frac{z_q \Delta a_j(x_q,y_q,z_q)}{\pi(x_q^2 + z_q^2 + 1)^2}. \tag{4.8}$$

The pixel area, $\Delta a_j(x_q,y_q,z_q)$, is normally equal to a constant for all $1 \le q \le n$. Computing all ΔF_q on the fly can be very expensive because a large number of pixels are involved and the computation is repeated for each pair of patches. Fortunately, this problem can easily be solved. Since the delta form-factor for each pixel is independent of any patch, all the ΔF_q can be pre-computed and stored in a table for later use. A table look-up technique is used to index all ΔF_q, and then they are summed to form the form-factor F'_{ij} in (4.5), which is equal to F_{ij}.

To sum up, for the form factors from a patch A_i to the rest of patches, the computation is divided into three steps. First, all the patches are projected onto each side of the hemi-cube centered on A_i, using an algorithm similar to the z-buffer algorithm. In the z-buffer, the frame buffer is replaced by an item buffer in which each item records the ID of the visible patch at the corresponding pixel. Then, the delta form-factors at those pixels with the same patch ID number are accumulated. The values of delta form-factors are obtained by looking them up in the delta form-factor table. The total summation (for all five faces of the hemi-cube) is the form-factor from A_i to the patch associated with the ID.

4.3 Front-to-back Algorithm for Form-Factors

A standard hemi-cube algorithm consists of two parts. The first part performs visibility computation from a source patch to the remaining patches. By convention, the source patch is called the *emitting patch*, expressed by $\mathcal{P}_e$, because it emits radiant energy in calculating form-factors. The remaining patches are called the *receiving patches* notated by $\mathcal{P}_r$. The relationship between "*emitting*" and "*receiving*" is relative to the current pair of patches whose form-factor is being calculated. A patch could be an "*emitting*" under a pair of patches; on the other hand, it could be a "*receiving*" under another of pair of patches.

In the second part, the form-factor from $\mathcal{P}_e$, to one of the remaining patches,$\mathcal{P}_r$, is calculated by counting the delta form-factors in the area projected by $\mathcal{P}_r$ onto the hemi-cube. Determining the delta form-factors in front-to-back algorithm requires two data structures: the delta form-factor tables and the 1-bit mask table.

4.3.1 Delta Form-factors and 1-bit Mask Table

As required by the traditional hemi-cube technique, two 2-D arrays of delta form-factors are pre-computed and any form-factor between two patches can be calculated quickly using a table lookup technique. The two arrays are called the *delta form-factor tables*(***DFFT***). One of these arrays corresponding to the upward-face of the hemi-cube is used to

accumulate delta form-factors when patches are projected onto the upward-face of the hemi-cube. The other is used when patches are projected onto any of the side faces.

In addition to the ***DFFT*** data structure, another *1*-bit 2-D array with the same size as the upward array of ***DFFT*** is used to assist the visibility computation. The array is named the *mask table* (***MT***) that actually replaces the item buffer and z-buffer traditionally used. Note that only half of the ***MT*** is used for calculating delta form-factors on the sides of the hemi-cube. The mask table serves two functions. First, it records pixels that have been covered by some high priority patches. When the table is full of 1's, it is determined that no more patches need to be painted onto the hemi-cube. This point will be discussed later. Secondly, the mask table distinguishes invisible pixels from those that are visible. It guarantees correct accumulation of delta form-factors.

The two data structure, ***MT*** and ***DFFT***, are the key elements to gracefully integrate the *octant priority list* (***OPL***) into the hemi-cube. Initially, the entire table of ***MT*** is set to 0's. When a patch is painted on a face of the hemi-cube, the corresponding pixel that is *0* in the mask table is set to *1*. However, it is only at those pixels newly set to *1* in ***MT*** that the delta form-factors are accumulated. An $N \times N$ array ***FF*** is used to hold the accumulated result for each form-factor. For example, ***FF***$[i][j]$ holds the form-factor between $\mathcal{P}_i$ and $\mathcal{P}_j$ after the painting process to be described is completed. At each intermediate stage of the painting process, ***FF***$[i][j]$ records the partial form-factors that are equal to the sum of delta form-factors at the pixels having been processed. Note that the form-factor from the emitting patch $\mathcal{P}_e$ to a receiving patch $\mathcal{P}_r$ is computed right after $\mathcal{P}_r$ is projected onto the hemi-cube. Thus there is no need to use an item buffer as used in traditional radiosity methods to store the projection of $\mathcal{P}_r$.

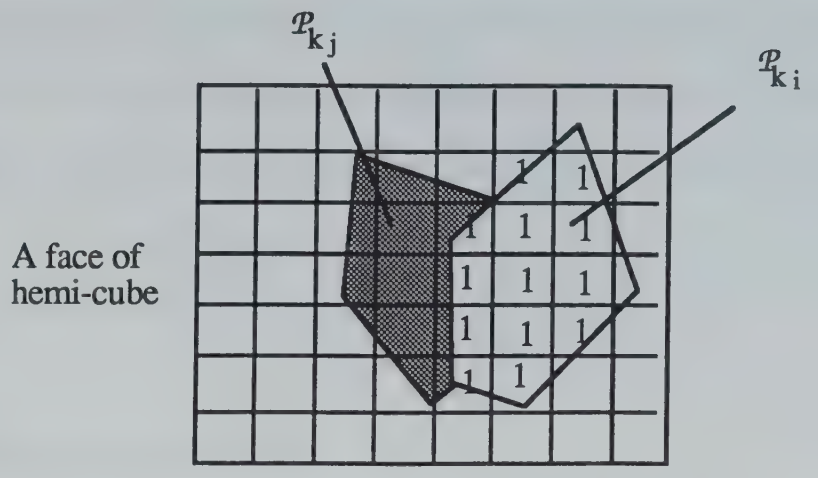


Figure 4.6. An example of painting process. When $\mathcal{P}_{kj}$ is partially occluded by $\mathcal{P}_{ki}$, only those delta form-factors within the visible area (the shaded part) of $\mathcal{P}_{kj}$ are counted.

Suppose that O_{ki} and O_{kj} are two octants in ***OPL*** ($i<j$), each of which contains a single patch, $\mathcal{P}_{ki}$ and $\mathcal{P}_{kj}$, respectively (for the more-than-one patch cases, the Newell-Newell-Sancha algorithm is used to determine the depth order of these patches), and that $\mathcal{P}_{ki}$ partially hides $\mathcal{P}_{kj}$ ($i<j$). Then only part of $\mathcal{P}_{kj}$ may be visible to the viewer. First, $\mathcal{P}_{ki}$ is projected and its pixels are set to *1* in ***MT***. When $\mathcal{P}_{kj}$ is painted, since its overlapping area with $\mathcal{P}_{ki}$ (which is invisible) is already covered by *1*'s in ***MT***, the delta form-factors in this area are not accumulated. Therefore, only the delta form-factors in the visible area are counted. Figure 4.6 illustrates this point.

4.3.2 Painting Patches onto the Hemi-cube

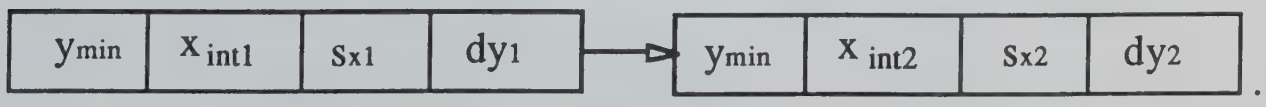
To project patches onto a face of the hemi-cube, two transformations are needed. One is the combination of translation and rotation that make the view point, the center of the emitting patch, coincide with the origin and the patch normal coincide with the z axis. The other is the perspective transformation with respect to the viewer and the plane onto which the patch is projected.

After these transformations, a 3-D patch is converted into a 2-D patch. A scan-line algorithm using an active edge table [FoDa83] is used to locate the pixels covered by the 2-D polygon onto a face of the hemi-cube, and their delta form-factors are then accumulated. In this algorithm, each edge of the polygon is represented by an edge structure as follows:

where

ymin — minimum y value of edge.
xint — x-coordinate of intersection of current scanline with edge.
sx — slope in x-direction.
dy — the vertical length of edge.

The edge structure is initially placed at the appropriate position in an *edge bucket* according to its minimum *y*. An *active edge table* (***AET***) contains a link list of those edges that are intersected by the current scan line. For example, when the current scan line is equal to *y*, as shown in Figure 4.7, the ***AET*** includes the two edges entries: e_1 and e_2.



To paint pixels between x_{int1} and x_{int2}, scanline conversion needs to be executed. Unlike the conventional scanline conversion that paints each pixel between x_{int1} and x_{int2} into the frame buffer [FoDa83, Ro85], the new method accumulates the delta form factors in the *delta form factor table* between the two columns that happen to correspond to x_{int1} and x_{int2}. Let $col_1 = x_{int1}$, $col_2 = x_{int2}$, ***DFFT*** be the pre-computed *delta form-factor table* corresponding to the upward-face of the hemi-cube and ***MT*** be the mask table. For any patch, the central part of the painting technique is given in the following pseudocode.

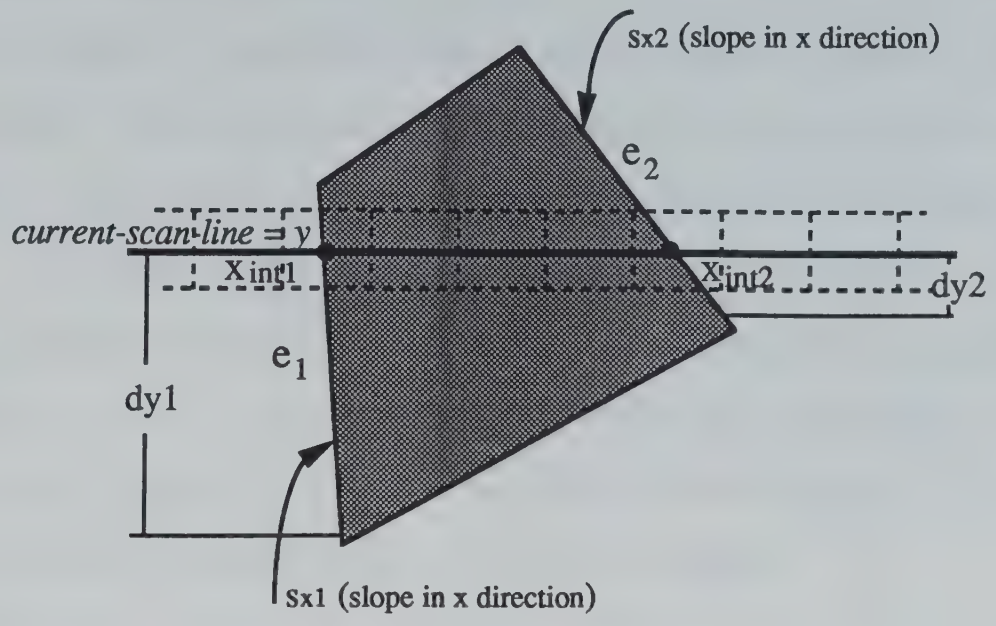


Figure 4.7. Scan-conversion. A scan line intersects with a pair of edges.

For each current-scan-line between the maximum *y* and the minimum *y* of the patch;

```
for(col=col1; col≤col2; col++) {
    if(MT[current-scan-line][col] == 0){
        FF[i][j] += DFFT[current-scan-line][col];
        MT[current-scan-line][col] = 1;
    }
}
```

After the above algorithm is repeated for each face of the hemi-cube, the final result of $FF[i][j]$ is the form-factor from the emitting patch $\mathcal{P}_i$ to the receiving patch $\mathcal{P}_j$. Note that the delta form-factor table for the upward-face is different from the one for the side-face, and normally these tables contain only the values in the upper-right quarter of the face because of symmetry. These factors are considered in the algorithm's implementation.

4.3.3 Why Front-to-Back?

Painting patches from front-to-back has a substantial advantage over a back-to-front painting scheme. This approach can save computation time, especially for a crowded environment. According to the painting scheme, when the mask table is full of *1* 's[1], it is obviously unnecessary to compute the form-factors from the emitting patch to the remaining patches that have not been processed. This is because these remaining patches have lower priority and are completely hidden by the patches that have been painted onto the hemi-cube. Therefore, their form-factors are determined to be zero. This situation happens very frequently in an environment with crowded objects.

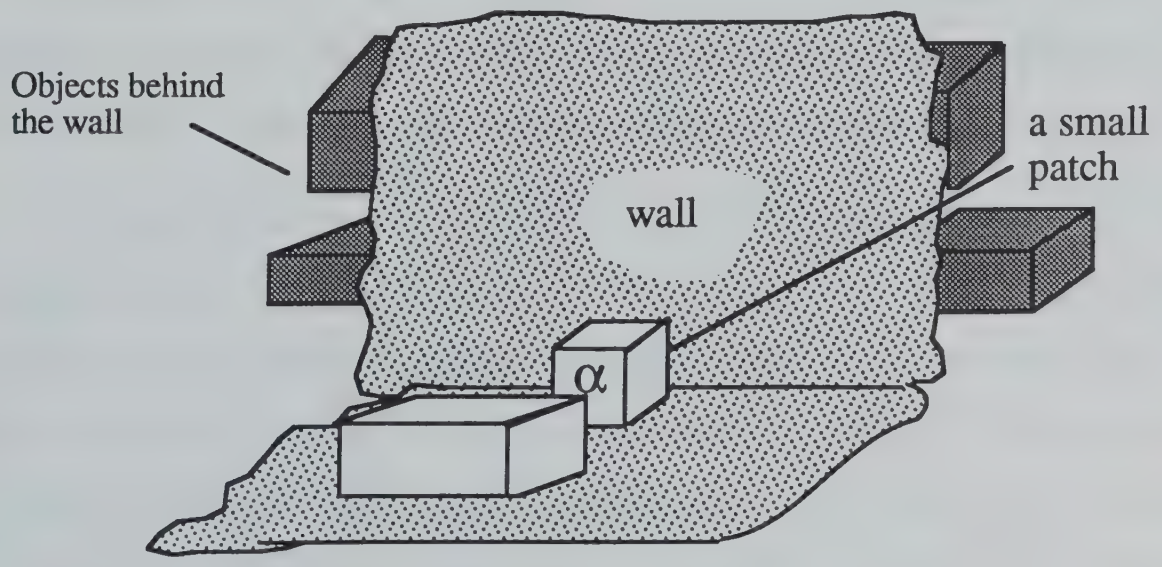


Figure 4.8. An example of zero form-factors. The small patch on the cube α can only see the wall that it faces. In other words, other objects receive no light from it.

Consider the example in Figure 4.8. A big wall or polygon appears in front of a small patch. All that the small patch can see is part of the wall (perhaps, a few other polygons around the boundary of the wall). Thus only the wall receives light from the patch. This implies that the form-factors from the small patch to any polygon other than the

[1] A counter, which counts the number of the pixels with 1, can be used to determine the fullness of 1's in the mask table.

wall are zero. As a result, in this case, only the big polygon (the wall), is projected and the delta form-factors only covered by the projection are accumulated, simply assuming that the form-factors to the other patches are zero.

In addition to saving computation time, the memory space can also be reduced because of the front-to-back characteristics. Another advantage is the elimination of the z-buffer and the item-buffer. Normally, a traditional hemi-cube uses two arrays with 200^2 floating point numbers for the z-buffer and 200^2 integers for the item buffer. The sizes of these arrays are strictly dependent on the tradeoffs between efficiency and accuracy for the form-factor computation. Low resolution of the hemi-cube will lead to serious aliasing problems, while high resolution will increase the demand of memory and slow down the form-factor computation. With the front-to-back algorithm, the resolution of the hemi-cube can be relatively large to achieve high accuracy without increasing the space problem.

Furthermore, the front-to-back algorithm allows anti-aliasing to be introduced in the radiosity method. It is difficult to use a z-buffer to remove various aliasing phenomena due to its brute-force technique. However, the front-to-back algorithm can use the octant priority information to determine the visible portion of patches within a single pixel and obtain more accurate delta form-factors for these patches. To accommodate anti-aliasing, the 1-bit mask table needs to be extended to a fraction mask table to record visibility information of patches inside a pixel. This anti-aliasing technique will be addressed in detail in the next chapter.

Other advantages of the front-to-back algorithm become clearer if the following factors are taken into consideration. Firstly, if it can be decided quickly whether a patch will be completely painted in the area already covered by all 1's in the mask table, the patch can immediately be rejected before it is actually painted onto the hemi-cube. Since octants are in spatial order when the octree is completed, it is not difficult to determine which patches are completely invisible using the information from the mask table and the octree.

Reynolds' dynamic screen technique can partially solve this problem [ReGo87]. Furthermore, if a patch is beyond a specified range, that is, the patch's minimum z is larger than a specified threshold, the patch should also be ignored. It is observed that a less-reflected and less-emitted object contributes light only in a local area. For example, a small and dull object would illuminate a smaller region than the light source or a polished surface. Therefore, it can be assumed that those objects that are far away from the dull object receive little or no light from the dull object, and that the form-factors to these far objects are zero.

Coherence has been used in computer graphics to improve efficiency. It has been discovered that form-factor computation has potential in making use of coherence of octants. Since octant priority is only dependent on the octant to which the viewer belongs instead of its absolute location, octant priority remains the same no matter where the viewer is located in the octant, and no matter in which direction the scene is viewed from. As a result, the order in which the patches are painted onto the upward-face of the hemi-cube is identical to that of any side-face because the viewpoint at the center of the emitting patch is identical for each face of the hemi-cube. This implies that there is no need to re-sort the octants for other side faces if the octant priority for the upward-face has been formed. Furthermore, if octants behind the emitting patch are identified and removed from the operation of projecting patches onto the upward face, for any side-face, these octants can be ignored immediately. Similarly, any octant ahead of the left side of the hemi-cube must be behind the right side, and vice versa. One octant ahead of the front side is behind the rear side, and vice versa. Therefore, a number of octants behind one side of the hemi-cube can be determined and discarded effectively, and that can save time in sorting octants and painting patches.

4.3.4 Overview of the Algorithm

The following pseudo code outlines the new radiosity method using octant priority established in the 3-D subdivision. Since the algorithm for 3-D subdivision is already shown in Algorithm 2 in Section 3.3.3, the focus is on the visibility computation by using the front-to-back approach.

ALGORITHM 4 (Overview of **FBRM**)

```
FBRM()
{
/*Pre-process Polygons into subpolygons */
/* One Ξ-lit for each vertex of polygonal light.*/
ShadowClipper() ⇒ Ξ-lit={χ1, ..., χH};

/* Construct the linear octree, and */
/* subdivide polygons into N patches, based on Ξ and Ξ-lit's */
Subdivision3D() ⇒ FOL={Ok'1, Ok'2, ..., Ok'M}, P = {P1,...,PN};
for ( i=1; i<=N; i++ ) {
    /* Find the octant priority list with respect to the center of Pi */
    OctantSort() ⇒ OPL={Ok1, Ok2, ..., OkM};
    for ( j=1; j<=M; j++ ) {
        if ( MT is full ) goto LOOP;
        Use Newell-Newell-Sancha algorithm to sort patches in Pkj ⇒ {Pj1,...,PjL};
        for ( l=1; l≤L; l++ ) {
            Project Pjl onto each face of the hemi-cube in turn ⇒ P'jl;
            Scan P'jl from max_y to min_y of P'jl.
            for ( current-scan-line=max_y; current-scan-line≥ min_y; current-scan-line-- ) {
                Calculate the intersections of current-scan-line with P'jl ⇒ min_x, max_x;
                Convert min_x ⇒ col1, max_x ⇒ col2;
                for ( col=col1; col≤col2; col++ )
                    if ( MT[current-scan-line][col] == 0 ) {
                        FF[i][jl][1] += DFFT[current-scan-line][col];
                        MT[current-scan-line][col] = 1;
```

---

[1] The subscripts are a little different from that in the piece of code shown on Page 76. The reason is that Algorithm 4 considers the cases of octants that contain more than one patch. The piece of code on Page 76 assumed that only one patch is in an octant. This assumption is just for illustration purpose.

```
                    }
                }
            }
        }
    LOOP:
}
```

Solve the radiosity equation to obtain: $B[1],...,B[N]$;
Render images using $B[1],...,B[N]$;

MT — mask table initialized to 0,
FF — form-factor matrix initialized to 0,
DFFT — delta form-factor table.

Chapter 5

Anti-aliasing for Form-factors

As seen in Chapter 4, form-factors are obtained by projecting patches on the hemi-cube. Since the hemi-cube is discretized into a grid of pixels, aliasing will always be present during the conversion from a patch to pixels. The aliasing problem will cause computation errors for form-factors, which directly affects the quality of images. First of all, this chapter examines the aliasing problems occurring in form-factor computation. A new concept is then introduced, known as the *fraction mask table* that is an extension of the *1-bit mask table* technique discussed in Chapter 4. Finally, an algorithm for anti-aliasing radiosity is discussed in detail.

5.1 Problems

Aliasing is a serious problem in image rendering. When an image display has a lower resolution than the image itself, the aliasing phenomena takes place. Generally, the aliasing phenomena is classified into three categories: jagged edges, small shapes popping on and off in time, and fine detail breaking up. These aliasing symptoms, particularly the staircasing edges, can be very noticeable as both quality and complexity of images increase.

As stated in Chapter 2, the derivation of form-factors is based on a pure mathematical model in thermal engineering. Why does the aliasing happen? If modern technology is able to perform polygon visibility, and double integration accurately and fast enough, the form-factor computation would become trivial and alias-free. Unfortunately,

researchers have only come up with approximate methods for a computer implementation of form-factor computation. The hemi-cube is one of the approximate methods. Basically, it employs a process of rendering images that inherits the common aliasing phenomena. Therefore, the aliasing problem indeed exists in computing form-factors. The occurrence of aliasing has an impact on the accuracy of form-factors that in turn causes incorrect solutions for intensity, and affects the quality of the final image.

A further observation indicates that aliasing in radiosity can be more serious than in other rendering approaches. In fact, the calculation of form-factors involves the image rendering process not only once, but N times, where N is the number of patches in the environment. Furthermore, the computation error caused by aliasing at each rendering process can be amplified dramatically.

Figure 5.1 shows a comparison between radiosity images with aliasing problems and without aliasing. The two pictures clearly reveal the difference between the two kinds of images. The image without aliasing has a smooth change of the reflected light from the ground, while its counterpart shows black-white interval strips on the ground. Note that the pictures only show the flat surfaces composed of regular-shaped patches. When the polygon meshing becomes irregular, the aliasing problem could be more noticeable. In order to demonstrate the aliasing problem, the images in (a) and (b) are shaded deliberately by using the constant shading method for each patch so that irregular pattern of intensity can be seen clearly. A final radiosity image, as shown in (c), however, requires a shading similar to Gouraud shading. This shading method determines each vertex intensity by averaging intensity values of surrounding patches and then interpolates intensity along patch edges and scanlines. Since this is a standard method [CoGr84] for displaying radiosity images, this thesis will not discuss it in any detail.

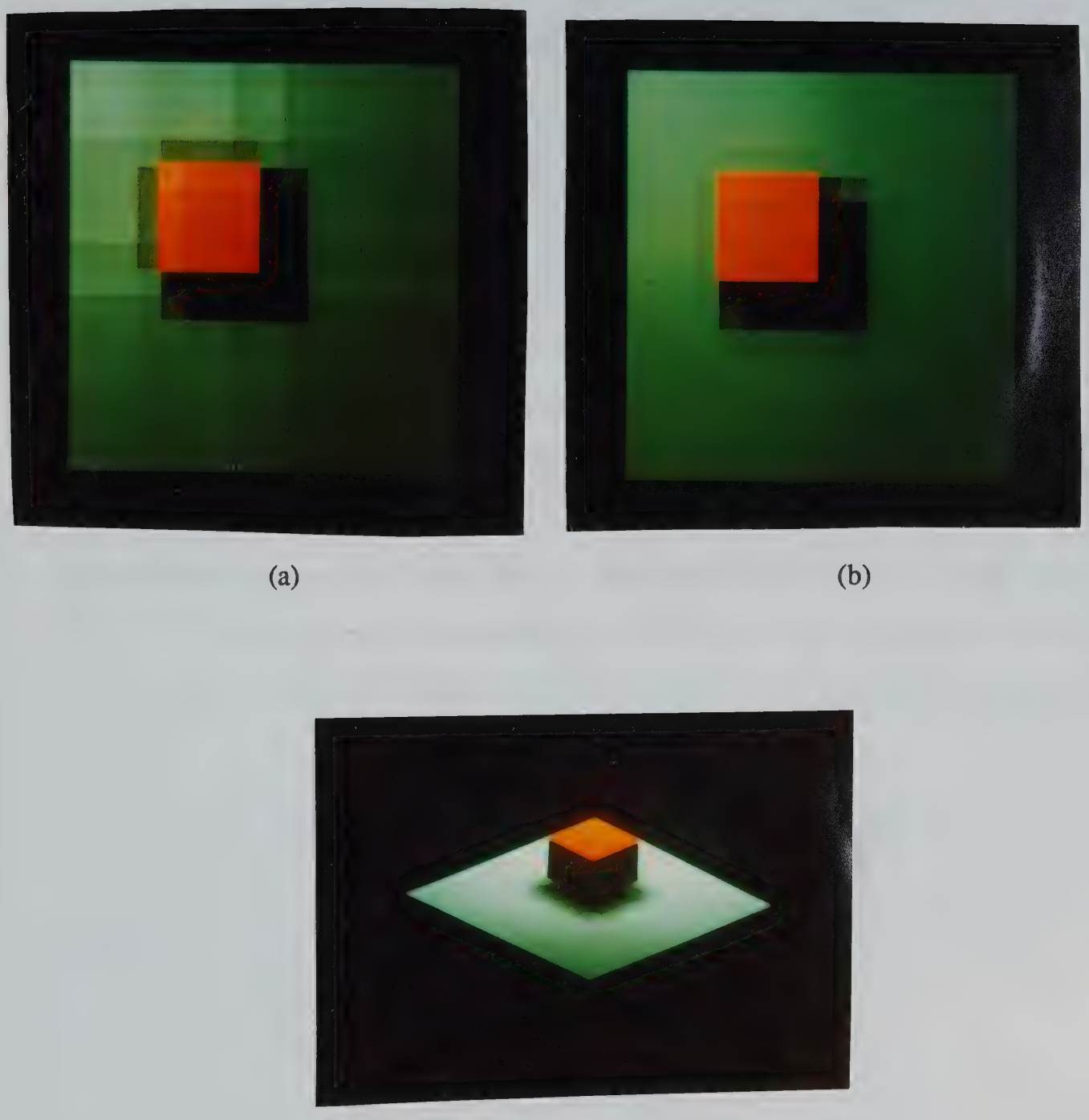


Figure 5.1. Comparison between images with aliasing and without aliasing. (a) Aliasing occurs in a form of black-white interval strips on the ground. This image is done by the algorithm described in the previous chapter. (b) Smooth change of intensity across the patches and the symmetric pattern of intensity on the ground indicate elimination of aliasing. This image is done by the algorithm described in this chapter. (c) The image with anti-aliasing (for both form-factors and final image) is shaded by the standard shading used in the hemi-cube method.

5.2 Area-weighted Approaches to Anti-Aliasing

There are many anti-aliasing methods for image rendering. The aliasing phenomena in radiosity takes place during two processes. One is the rendering process that directly causes aliasing artifacts on the final image. The other is the form-factor calculation. To understand this, regard the process of calculating form-factors from an emitting patch to the remaining patches as shooting a picture at the center of the emitting patch through a side of the hemi-cube. Thus, form-factor calculation is equivalent to the rendering process. Catmull's area-weighted anti-aliasing [Catm78] or Carpenter's A-buffer method [Carp84] can then be used directly to add an alias-free feature to form-factor calculation. To take advantage of the front-to-back algorithm, this thesis develops an alternative technique to tackle the aliasing problem. The following discussion on anti-aliasing is focused on the form-factor calculation instead of the rendering. The same ideas, however, are suitable for rendering.

An area weighted approach to alias-free image generation integrates the intensities over the visible areas at a single pixel. If a pixel is covered by three areas, a_1, a_2 and a_3 with different intensities, I_1, I_2 and I_3, respectively, as shown in Figure 5.2, the intensity at the pixel is equal to the average intensity:

$$I = \frac{1}{A}\int_{y1}^{y2}\int_{x1}^{x2} I(x,y)dx\ dy$$

$$\doteq \frac{1}{A}\sum_{i=1}^{3} I_i a_i \qquad (5.1)$$

where A is the area of the pixel.

In the case of form-factor calculation, the area-weighted approach is slightly different because the important aspect of each pixel is the form-factor rather than the intensity. In fact, the form-factor at a pixel is a delta form-factor that represents the fraction

of energy leaving the emitting patch to the patches covering the pixel. Traditionally, when the receiving patch partially covers the pixel, the delta form-factor is approximately obtained by assuming that the whole pixel is covered. This is where the aliasing originates.

To resolve the aliasing problem, a fraction of the delta form-factor should be considered. Unlike a method for area weighted intensity that sums up the weighted intensity, the delta form-factor at each pixel is broken down into smaller delta form-factors, each of which is assigned to the corresponding receiving patches covering the pixel.

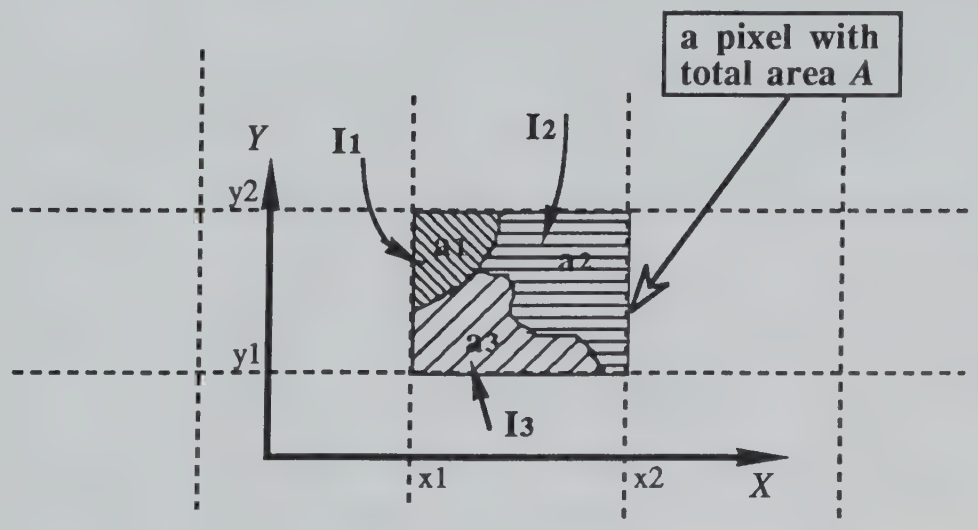


Figure 5.2. Average intensity. The intensity at a pixel is equal to the average intensityof I_1, I_2 and I_3.

Consider the example shown in Figure 5.2. Let a_i be the area covered by patch $\mathcal{P}_i$ where $i = 1, 2$ and 3. Let ΔF represent the delta form-factor for this pixel. ΔF is obtained by indexing ***DFFT*** (the *delta form-factor table*). Thus, the delta form-factor ΔF_i between the emitting patch and $\mathcal{P}_i$ is calculated by

$$\Delta F_i = \frac{a_i}{A} \times \Delta F \qquad (i = 1, 2, 3) \tag{5.2}$$

The correctness of the equation becomes evident if the hemi-cube is discretized into more pixels. Referring to Section 4.2, it is known that the form-factor between two

patches is obtained by accumulating all the delta form-factors on pixels covered by the receiving patch. Since a pixel has area, each delta form-factor can be regarded as the form-factor from the emitting patch to the corresponding pixel. In addition, a form-factor is proportional to the area of the receiving patch. That is, a bigger patch receives more light than a smaller one. As a result, when a patch only occupies part of a pixel, it is evident that the form-factor to the occupied portion of the pixel should be a fraction of the form-factor to the whole pixel. This observation results in the weighting system for delta form-factors - each delta form-factor will be weighted with the fraction of the occupied area of the whole pixel area.

TABLE 2 (Seven Basic Classes)

Classes	Shapes	Characteristics
C_1		*right side is vertical; bottom is horizontal*
C_2		*right side is vertical; top is horizontal.*
C_3		*rectangular*
C_4		*left and right are vertical; top and bottom have slopes up.*
C_5		*left and right are vertical; top and bottom have slopes down*
C_6		*left side is vertical; bottom is horizontal*
C_7		*left side is vertical; top is horizontal*

5.3 Weighted Delta Form-factors

To introduce the weighted delta form-factors into **FBRM**, only the inner loop in Algorithm 4 needs to be modified. The segment between a pair of active edges is decomposed into several basic parts shown in Table 2.

5.3.1 Decomposition of Segments

Traditionally, a patch is scanned line by line into pixels on the hemi-cube. With the consideration of anti-aliasing, a patch is decomposed into a sequence of segments (having a certain thickness or area), each corresponding to a scanline between the top and the bottom of the patch. Then each segment is decomposed again into parts along the x direction. In order to calculate the area of each part efficiently, the decomposition algorithm ensures that each part belongs to one of the seven categories in Table 2.

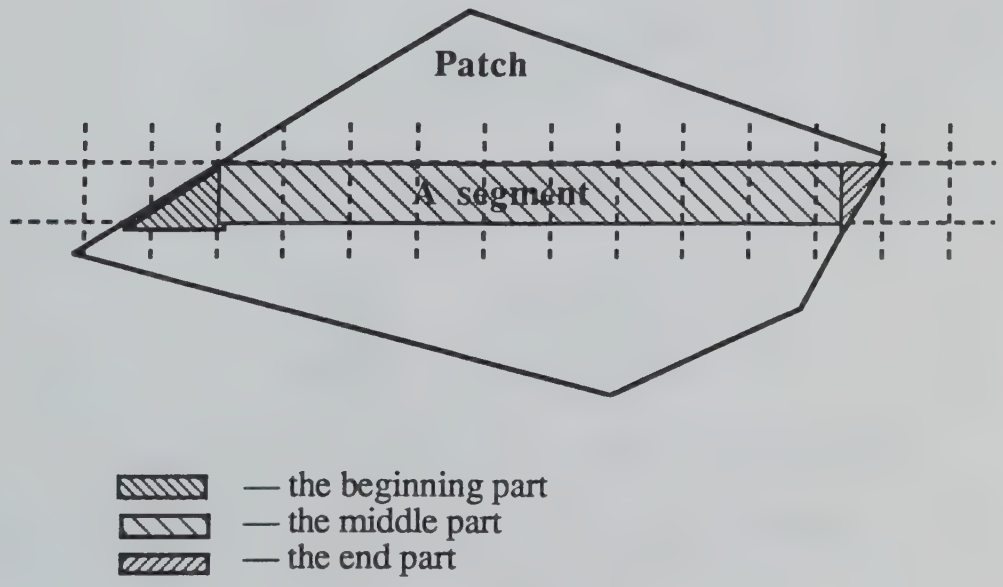


Figure 5.3. An example of a segment and its three parts.

Normally, a segment can be divided into three parts: the beginning, the middle, and the end, as shown in Figure 5.3. Generally, the beginning part belongs to one of the

first two classes, C_1 and C_2. The middle belongs to one of C_3, C_4, C_5. The end belongs to one of C_6 and C_7. A segment may not necessarily be decomposed into all the three classes. It could be decomposed into any combination of the three. For example, segment *A* in Figure 5.4 is composed of a beginning and an end. Segment *B* consists of only the middle part (C_4). Segment *C* is made up of a middle and an end.

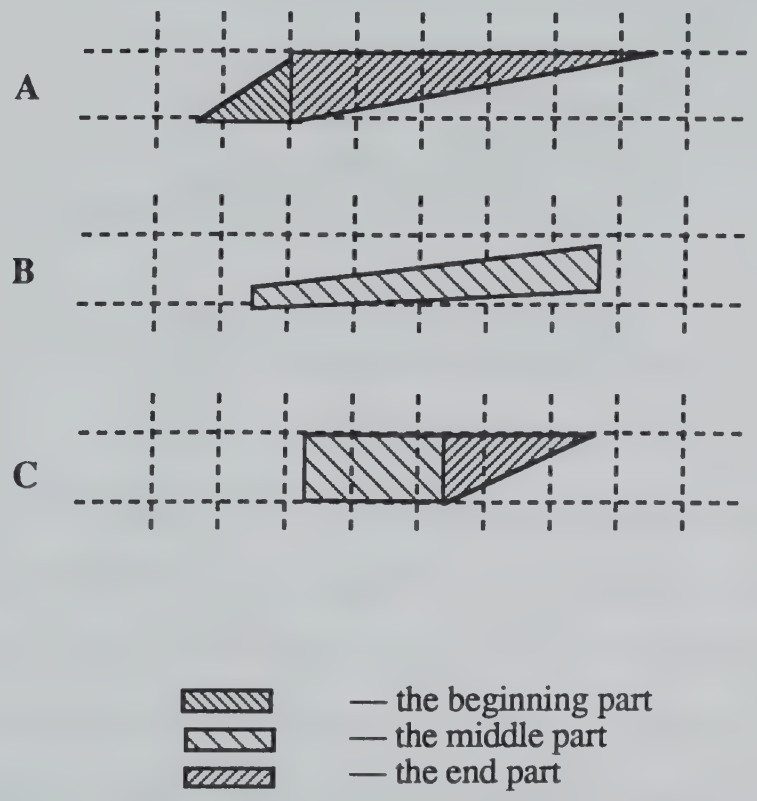


Figure 5.4. Examples of the segment decomposition.

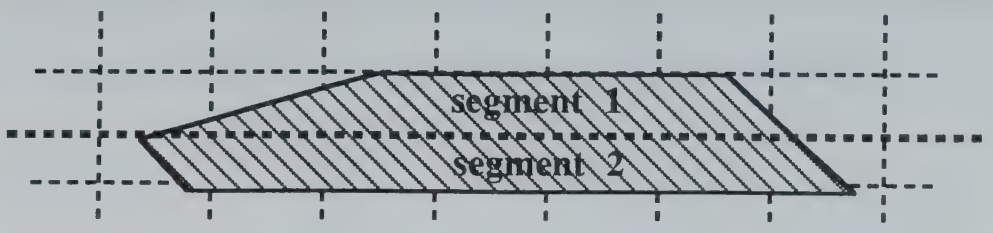


Figure 5.5. Failure of segment decomposition.

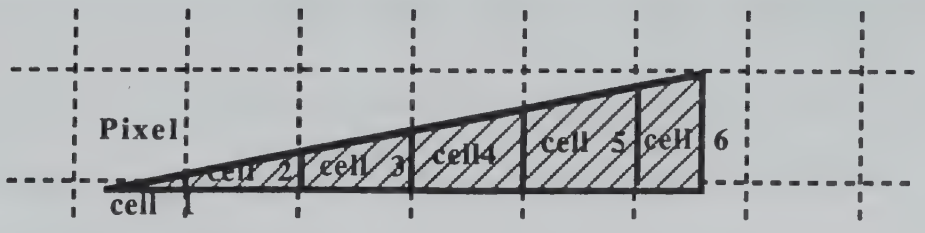


Figure 5.6. Cell areas. A cell area is determined by intersection between the boundary of the part and the current pixel.

Note that if the segment defined here cannot be decomposed into these parts, the segment needs to be broken into more segments in the *y* direction. Figure 5.5 shows an example of this case. Obviously, in the example, the beginning part cannot be classified into one of the seven classes. To resolve this problem, the segment is divided into two segments along the dotted line. After the operation, each of the two segments becomes a decomposable segment.

After a segment is decomposed into three parts, the painting process is applied to each of them. The painting process varies for different parts. In the positive x direction for each part of the segment, cells are determined by calculating the intersection of the boundary of the part with the boundary of the current pixel, as shown in Figure 5.6.

It is interesting to notice that each cell is a regular geometry, a trapezoid[1]. Therefore, the area of each cell is calculated using a uniform formula.

$$\textbf{\textit{cell\_area}} = 0.5 \times (\textbf{\textit{left\_ht}} + \textbf{\textit{right\_ht}}) \times \textbf{\textit{width}}. \quad (5.3)$$

For each cell in class C_3, it is easy to calculate (5.3) since ***left_ht*** = ***right_ht***. In fact, (5.3) becomes the area formula for the rectangle. The value of ***width*** can be the width of a pixel, or the distance between the far left edge of the middle part and the left boundary of the second pixel that it covers, or the distance between the right boundary of the second last pixel and the far right edge. This point can be seen in Figure 5.7.

For the beginning part, the cells can be a trapezoid or a triangle. To determine the areas of the cells, four variables are needed: *x_left*, *x_right*, *y_top* and *y_bottom*. *x_left* and *x_right* are incremented by a pixel. One of *y_top* and *y_bottom* remains constant depending on the slope of the beginning part. In the first case of Figure 5.8, *y_top* is constant and *y_bottom* is updated by *dy* (= *slope*pixel_width*). Then the area for each cell is obtained by calculating (5.3) with

left_ht = *y_top - old_y_bottom*,
right_ht = *y_top - new_y_bottom*,
width = *x_right - x_left*.

For the second case in Figure 5.8, the roles of *y_top* and *y_bottom* are swapped. Thus,

left_ht = *old_y_top - y_bottom*,
right_ht = *new_y_top - y_bottom*,
width = *x_right - x_left*.

Similarly, area formulas for the end class can also be derived.

The remaining classes are the strips. Their cell areas are more difficult to calculate because none of *y_top* and *y_bottom* stay constant. Both *y_top* and *y_bottom* are updated

---

[1]Triangle or rectangle is the special case of trapezoid

according to the corresponding slopes. Thus each cell area in class C_4 or C_5 is calculated based on the following values (see Figure 5.9):

left_ht = *old_y_top - old_y_bottom,*
right_ht = *new_y_top - new_y_bottom,*
width = *x_right - x_left.*

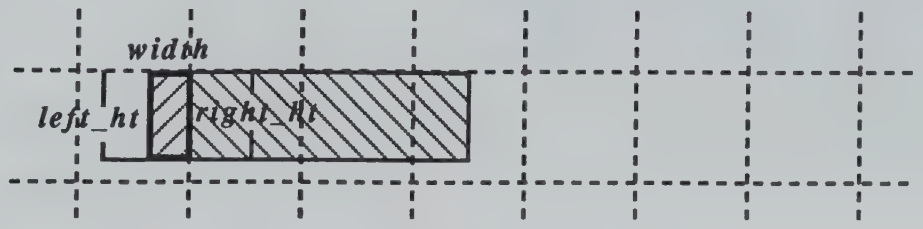
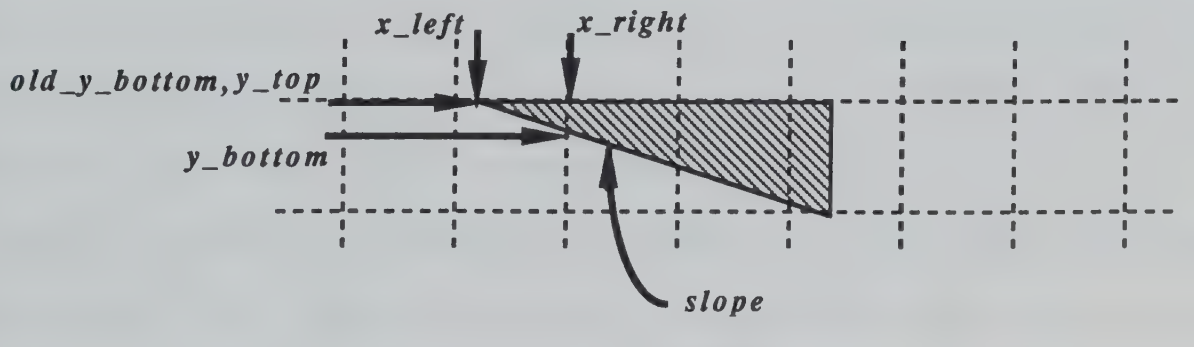
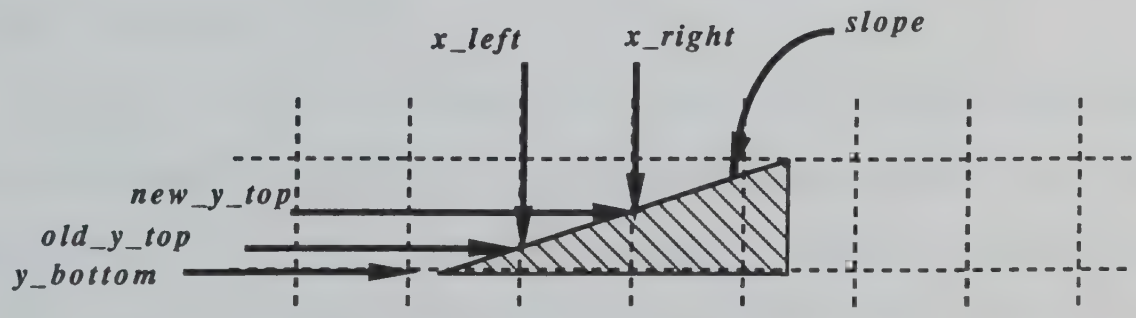


Figure 5.7. Calculating cells in C_3.



Case 1



Case 2

Figure 5.8. Calculating cells in C_1 and C_2.

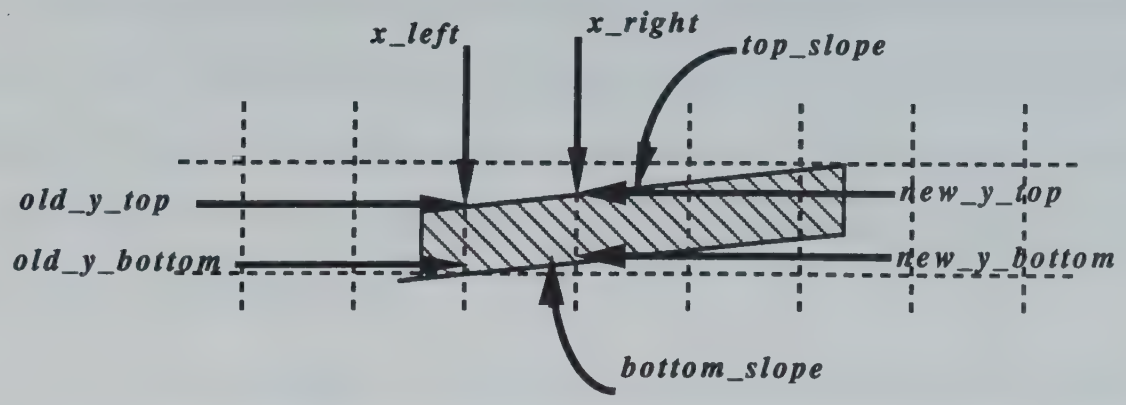


Figure 5.9. Calculating cells in C_4 and C_5.

5.3.2 Fraction Mask Table

The values of cell areas are used to determine the fraction of coverage of a receiving patch at a pixel by dividing it by the total area of a pixel. The fraction number is then used as the weight of the delta form-factor. It is easy to determine the weight if a pixel is occupied partially by a single visible patch. However, in most situations, things are not always this simple. The visibility computation and the weight calculation need to be considered together when more than one patch occupy a single pixel. In this case, the cell-area given by (5.3) to (5.6) cannot be converted directly into the weight. In order to add visibility factors to the weights, the *1-bit mask table* is extended to a 32-bit mask table called *fraction mask table*, or ***FMT***.

Suppose $\mathcal{P}_i$ and $\mathcal{P}_j$ partially occlude a pixel, where $\mathcal{P}_i$ has a higher priority than $\mathcal{P}_j$. Thus $\mathcal{P}_i$ is first painted to cover the pixel. Conventionally, the coverage of the pixel is 100%, no matter how $\mathcal{P}_i$ actually occludes the pixels, and $\mathcal{P}_j$ is regarded as invisible at that pixel. In the case of anti-aliasing, $\mathcal{P}_i$ is considered to be partially visible and only a fraction of the pixel occluded by $\mathcal{P}_i$ is calculated. $\mathcal{P}_j$ may become visible at that pixel too. The question is: what part of $\mathcal{P}_j$ is visible at the pixel? In other words, what percentage of $\mathcal{P}_j$ covers the remaining visible portion of the pixel?

Ideally, recording the edge positions of $\mathcal{P}_i$ that intersect with the boundary of a pixel can answer this question, but it is too expensive in terms of computation. Alternatively, for each pixel, a fraction is used to approximate the coverage of the area that has been occupied by the higher priority patches. An algorithm is then applied to adjust the weight of lower-priority patches.

Let F_m (<=1) represent the coverage of a pixel by the higher-priority patches, say $\mathcal{P}_i$. Let F_n be the weight for the lower-priority $\mathcal{P}_j$ calculated by one of Equations 5.3 to 5.6. When $\mathcal{P}_j$ is processed, the adjusted weight for $\mathcal{P}_j$ is calculated according to the following situations.

CASE 1 if F_m is equal to 1, the weight is equal to 0 and F_m remains unchanged.

CASE 2 if F_m is equal to 0, the weight is equal to F_n and F_m is changed to F_n.

CASE 3 If $F_n > 1\text{-}F_m$, $1\text{-}F_m$ is assigned to the weight and F_m is adjusted to 1, which means that the pixel is occluded completely.

CASE 4 If $F_n <= 1\text{-}F_m$, the weight is hypothetically F_n and F_m is changed to F_m+F_n.

In the case of no overlap, CASE 3 and CASE 4 yield correct weights. In the case of partial overlapping, if the lower priority patch is not completely hidden by the higher priority patch, CASE 3 and CASE 4 can still give correct or approximate results. The situation shown in Figure 5.10, however, produces incorrect weights. This case will be discussed in detail in the last paragraph of this chapter.

In these four cases, updating F_m properly is important to tracking correct visibility of patches. Replacing ***MT*** in Algorithm 4 of Chapter 4, ***FMT*** is used to record F_m's. Each element in the new table is a floating-point number with 32-bits. The modified algorithm is shown as follows.

ALGORITHM 5 (Anti-Aliasing **FBRM**)

```
FBRM()
{
/*Pre-process Polygons into subpolygons */
 /* One Ξ-lit for each vertex of polygonal light.*/
ShadowClipper() ⇒ Ξ-lit={χ1, ..., χH};

/* Construct the linear octree, and */
/* subdivide polygons into N patches, based on Ξ and Ξ-lit's */
Subdivision3D() ⇒ FOL={Ok'1, Ok'2, ..., Ok'M}, P = {p1,...,pN};
for ( i=1; i<=N; i++ ) {
    /* Find the octant priority list with respect to the center of pi */
    OctantSort() ⇒ OPL={Ok1, Ok2, ..., OkM};
    for ( j=1; j<=M; j++ ) {
        if ( MT is full ) goto LOOP;
        Using Newell-Newell-Sancha sort patches in Pkj ⇒ {P'j1,...,P'jL};
        for ( l=1; l≤L; l++ ) {
            Project P'jl onto each face of the hemi-cube in turn ⇒ P'jl;
            /********************* Modified Part ************************/
            Scan P'jl from max_y to min_y of P'jl.:
            for( y=max_y, current-scan-line=0; y>=min_y; y--, current-scan-line++ ) {
                obtain a new segment ⇒ Sy
                for ( col=col1; col≤col2; col++ ) {
                    Calculate cell-area/pixel_area ⇒ Fn  /*for different parts, the
                                                          calculation is different*/
                    if ( FMT[current-scan-line][col] == 1 ) Fm = 0;
                    else if ( FMT[current-scan-line][col] == 0 ) Fm = Fn;
                    else if ( Fn > 1 - FMT[current-scan-line][col] )
                        Fm = 1 - FMT[current-scan-line][col];
                    else
                        Fm = Fn
                    FF[i][jl] += Fm×DFFT[current-scan-line][col];
                    FMT[current-scan-line][col] += Fm;
                }
            }
            /******************* Modified Part End ************************/
        }
    }
    LOOP:
```

}
Solve the radiosity equation to obtain: $B[1],...,B[N]$;
Render images using $B[1],...,B[N]$;

F_m — the weight of delta form-factors,
F_n — cell-area/pixel_area,
FMT — fraction mask table initialized to 0,
FF — form-factor matrix initialize to 0,
$DFFT$ — delta form-factor table.

There are two problems that need to be pointed out. The first problem is the overhead with the anti-aliasing operations. The second is the accuracy of the anti-aliasing results. In the above algorithm, the most intensive parts of the computations lie on the decomposition of segments and the comparison of F_m and F_n. The decomposition operations become complex when a number of odd-shaped patches are involved. A odd-shaped patch normally produces a long beginning part and a long end part at the two ends of a segment. This case often happens around the edges of the patch. But generally speaking, the interior of a patch occupies most of the pixels, while the edges occupy a small portion. As a result, a patch is normally decomposed into more of the third elementary parts (the rectangle) than the others. The experimental results show that more than 80% of decomposed parts are rectangles. Since the area of a rectangle requires less time to calculate, the overhead is relatively small.

As far as quality of images is concerned, speed has to be sacrificed a little. The question is how much is gained by improving image quality. As mentioned previously, F_m only approximately represents the visibility of higher priority patches at a pixel. It has no clue about the exact locations of the coverage of a pixel by higher priority patches. All calculations are based on the pure percentage of the coverages. Therefore, the visibility determination and the weight calculation would be incorrect in the situation shown in Figure 5.10. According to Case 4 of the algorithm, the part of the pixel occupied by the lower priority patch $\mathcal{P}_j$ is regarded to be visible because $F_n \leq 1\text{-}F_m$. F_m is updated to F_m

(old)+F_n and the visibility fraction assigned to $\mathcal{P}_j$ is F_n. Apparently it is not correct because $\mathcal{P}_j$ is obscured by $\mathcal{P}_i$ completely. The miscalculation does create some aliasing in computing form-factors. Fortunately, the aliasing problem is not as serious as for the traditional hemi-cube. This is due to the fact that the kind of situations in Figure 5.10 are rare for most scenes. Since polygons are subdivided into patches, in most cases, two close patches are originally on a common polygon and thus share a common edge. The two patches have to be processed first before other lower priority patches are processed. The sum of the two patch's weights are equal to 1. Therefore, the visibility at the pixel is never miscalculated. The above problem takes place only in the case where the edges of two separated polygons happen to cross the same pixel and occlude the same part of this pixel. Apparently, it is not common. In fact, the image in Figure 5.1(b) has shown great improvement by the anti-aliasing method.

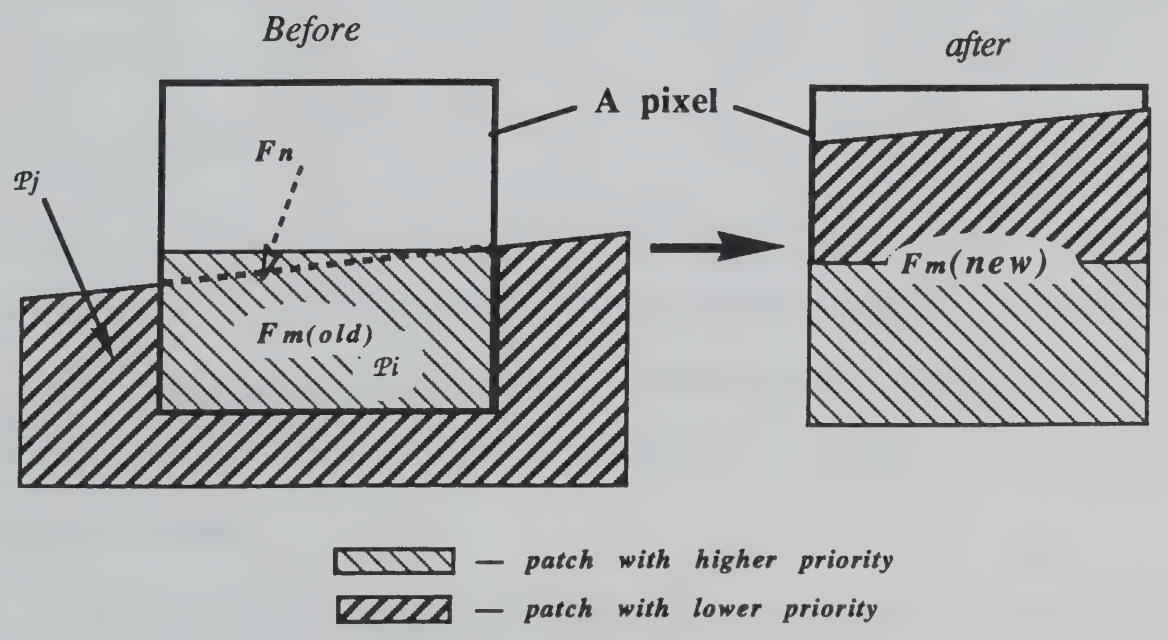


Figure 5.10. An example of incorrect visibility computation where the new F_m is incorrectly updated to F_m(old)+F_n. The correct new F_m should be is equal to the old F_m.

Chapter 6

Parallel Implementation of Radiosity

Software tends to reach its limitation under conventional hardware technology. Investigation on parallel implementation of radiosity is in the right direction in order to make radiosity techniques feasible. By an in-depth study and experiments, it is discovered that a radiosity method involves a high degree parallelism. This chapter explores potentials of parallel radiosity methods and gives some ideas of how to approach them. A simple parallel scheme based on the **FBRM** is described in detail. Other parallel methods are also given with algorithmic detail.

6.1 Introduction

As computing technology progresses, it becomes more and more clear that computer software built on the conventional computers with a single processor has reached its limits to accomplish sophisticated and complicated applications. Realistic image rendering is one of these applications. The solution to this problem is to invent more powerful and parallelized computers.

Generally speaking, there are two kinds of approaches to parallelize image-oriented computation: pipelining and parallel pixel-based processing. The pipelining scheme partitions the whole process into different components, each performing a specific operation by inputting data from the last component and outputting the resulting data to the next one. For the parallel pixel-based processing, the same computation is performed at

every pixel concurrently. These two schemes gain great speed-up on rendering and animating images.

Ray-tracing is a good candidate for the parallel scheme. A ray emitting from the eye is traced through each pixel into the 3-D space. The process is repeated for each pixel independently. Therefore, if each pixel has a corresponding processor that executes the ray-tracing process for that particular pixel, the speed-up will obviously be increased by the number of processors.

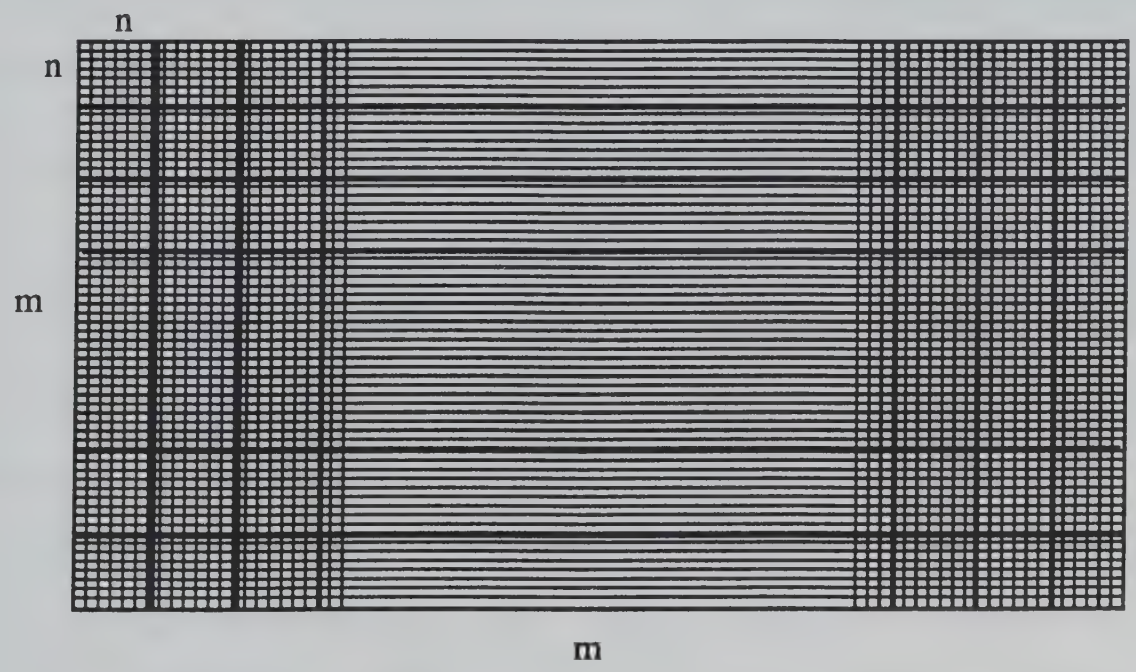


Figure 6.1. Partitioning of pixels. The pixels on the screen are partitioned into $m \times m$ groups each containing $n \times n$ pixels.

A single processor for each pixel will be very expensive in terms of hardware. However, grouping of several pixels for a single processor seems to be feasible and still faster than a sequential scheme. Thus, the screen can be partitioned into $m \times m$ equal-sized squares, each containing $n \times n$ pixels and corresponding to a processor, as shown in Figure

6.1. Each processor deals with its own square-shaped pixels by tracing rays through them one by one. With this parallel scheme, the computation is reduced by a factor of m^2.

Radiosity is more complicated than ray-tracing and other rendering techniques. Its parallelism exists at several levels and by different degrees. As seen in previous chapters, the bottle neck in the radiosity method is the form-factor computation. Without reducing its computation costs, any parallel algorithm applicable to other parts of the radiosity method has little contribution to speeding up this image method. In the following sections, a parallel scheme for computing form-factors is described. A parallel algorithm for solving radiosity equations is then introduced, and a more sophisticated approach is discussed at the end of this chapter.

6.2 Parallel Computation for Form-factors

To solve radiosity, $B_1,...,B_N$ (refer to (2.2)), the $N \times N$ form-factors, F_{11}, ..., F_{1N}, ..., F_{N1}, ..., F_{NN}, need to be determined. From the discussion and analysis described in the previous chapters, it is discovered that F_{ij} ($i,j = 1,...,N$) is computationally independent of one another. This implies a parallel solution like the pixel-based parallel scheme can be accommodated into form-factor computation. An ideal scheme could be having $N \times N$ processors, each calculating a form-factor. Although it is simple and inexpensive (almost $O(1)$ cost), this scheme is ineffective when N is a large number. Nevertheless, the grouping idea seems to work again for F_{ij} calculation.

Referring to (2.2), for each row of F_{ij}, i.e., F_{i1}, ..., F_{iN} (for each $i = 1,...N$), a tightly integrated process is executed to obtain all F_{ij} of this row at once. In fact, because of the visibility issue, it is impossible to merely calculate a single F_{ij} without considering the rest. It is a common case in the **FBRM** that some F_{ij}'s have to be determined before a particular F_{ij} can be calculated, because the particular F_{ij} may be partially visible. In other

words, during calculating this particular F_{ij}, some F_{ij}'s are obtained as side products. For simplicity, all of F_{ij} in a row are, therefore, grouped into the same processor.

Suppose that the radiosity program is run on a distributed system with H processors: PE_1, ..., PE_H. Let N be the total number of patches after the 3-D subdivision. The parallel processing for the N^2 form-factor F_{ij} is divided into five steps.

1. Each PE_i (i=1,..,H) inputs a copy of the entire modelling database, P={$\mathcal{P}_1$, ..., $\mathcal{P}_N$}. Note P is a set of all patches produced by the 3-D subdivision.

2. The N indices of {$\mathcal{P}_1$, ..., $\mathcal{P}_N$} are shuffled randomly, obtaining:

   $$i_1, \ldots, i_N.$$

   For simplicity, assume that {1, ..., N} is already shuffled.

   Then they are divided into H groups, each with $L=\lceil N/H \rceil$ elements

   $$g_1 = \{1, \ldots, L\}$$
   $$g_2 = \{L+1, \ldots, 2\times L\}$$
   $$\vdots$$
   $$g_H = \{L\times(H-1)+1 \ldots, N\}$$

   The random shuffling of indices is necessary to ensure that each processor has a similar load of tasks and terminates approximately at the same time. Of course, this trivial distribution method may not always guarantee this. It can, however, avoid grouping neighbor patches into a processor because the neighborhood patches tend to have close indices according to the 3-D polygon subdivision. Frequently, if a patch sees a few polygons in a scene, its neighbor patches see these same polygons. This means they take similar computational time to calculate their corresponding row of F_{ij}. Therefore, the result would be that some processors are very busy and take longer to finish,

while some have a lighter burden and finish very quickly. From the overall performance point of view, this is not effective for a parallel system.

3. At each process or PE_k (k= 1, ..., H), the following computation is performed.

   Let $g_k = \{k_1, k_2, ..., k_L\} = \{L\times(k\text{-}1)+1, ..., L\times k\}$.

   for I in g_k\{

   Determine the center of $\mathcal{P}_I$;

   Sort $\{\mathcal{P}_1, ..., \mathcal{P}_N\}$ with respect to the center using the octree;

   $\Rightarrow \{\mathcal{P}_{J_1}, ..., \mathcal{P}_{J_N}\}$;

   Paint $\{\mathcal{P}_{J_1}, ..., \mathcal{P}_{J_N}\}$ in order onto the hemi-cube and

   Calculate $F_{IJ_1}, ..., F_{IJ_N}$;

   \}

4. Output the partial matrix:

$$\begin{pmatrix} F_{k_1 1} & ..... & F_{k_1 N} \\ \vdots & & \vdots \\ F_{k_L 1} & ..... & F_{k_L N} \end{pmatrix}.$$

5. Merge the outputs from all PE_1, .., PE_H into a full $N\times N$ matrix:

$$\begin{pmatrix} F_{11} & ..... & F_{1N} \\ \vdots & & \vdots \\ F_{N1} & ..... & F_{NN} \end{pmatrix}.$$

Then output it to the next step: solving the radiosity equations.

6.3 Implementation on Myrias SPS-2

To experience parallel radiosity technology, the above simple scheme was implemented on the Myrias SPS-2 supercomputer designed and built by the former Myrias Research

Corporation. To understand the implementation, the following paragraphs give a brief introduction to the supercomputer.

The Myrias SPS series are Scalable Parallel Supercomputers for massively parallel processing, especially suitable for applications like scientific computations. The SPS-2 system hardware comprises a Master Controller, a number of Processor Cages and a Communication Network. The Master Controller includes an 8 MByte Sun 3/280™ processor with a 19" color monitor and other peripheral devices. It provides interfaces and networking to the end users or other workstations. Each of the Processor Cages contains up to 64 Motorola 68020 processors, with Motorola 68882 floating point units, and 256 MBytes of memory. The number of processors can be configured by users and the parallel tasks are evenly distributed among these processors by the Myrias Parallel Application Management System (PAMS). The processor arrangement is totally transparent to users. The Communication Network provides a hierarchical arrangement of parallel buses to make all of the system memory accessible to each processor.

The Myrias PAMS mainly assigns tasks to processors, balances the load between processors, and moves data between tasks automatically. PAMS also includes a Myrias Parallel C (MPC) compiler that provides an extra statement **pardo** for parallel programming. The Myrias MPC and the Myrias operating environment is externally similar to ANSI C and Berkeley Unix. Users compile their C code using mpc, and initiate the executable on the SPS-2 with an "mrun" command, specifying the number of processors selected for execution. With each execution of MPC code, a set of status data is printed on the console to indicate the memory utilization, user time, system time, page wait time, idle time, and the number of tasks started and running.

In our application, the front-end processor (Master-Controller) was used to accomplish the 3-D subdivision and shuffle the indices sets: g_1, ..., g_{64}. Then, the **pardo** statement is used to partition jobs into the 64 PE's:

```
pardo ( i from 1 to 64 ) {
    for( j=0; j<L; j++ ) {
        I = g_i[j];
        FF = FF1toN( p_I) ;
        write( FF ) (into file );
    }
}
```

where $L = \lceil N/64 \rceil$, N is the number of patches, and ***FF*** points to an array of a row of form-factors: F_{i1}, ..., F_{iN} that **FF1toN** calculates.

With this distributed scheme, the implementation results shows that the radiosity method, **FBRM**, is approximately 60 times faster. The status data reported by Myrias SPS-2 indicates that the user time is very impressive, almost 90%, and the system and idle time take only a very small portion of the total elapsed time. The speed-up of this parallel scheme is almost linear in the number of processors. Note that the parallelism is performed among separate rows of F_{ij}. To achieve a greater reduction of computation, parallelism embedded in calculating a single row of form-factors F_{i1}, ..., F_{iN} is worthwhile to be investigated. This issue will be addressed in Section 6.5.

6.4 Parallel Algorithm for Solving Radiosity Equations

Radiosity equations are a set of simultaneous equations with N radiosity unknowns. Normally, N is greater than 1000. It takes a noticeable amount of time to obtain accurate enough solutions. As pointed out previously, a Gauss-Seidel method is suitable for solving the equations. The sequential implementation of **FBRM** has used the Gauss-Seidel iteration. For a parallel algorithm specific to the SPS-2 Myrias with 64 distributed processors, however, the Jacobi iteration is used. Unlike the Gauss-Seidel that evaluates the new values of the unknowns using most up-to-date "old" values, the Jacobi iteration evaluates the next approximation of the unknowns based on the "old" values derived at the

last iteration. On the SPS-2, it is difficult to access the most up-to-date value that was computed on another processor. This is the reason that the Jacobi iteration is used.

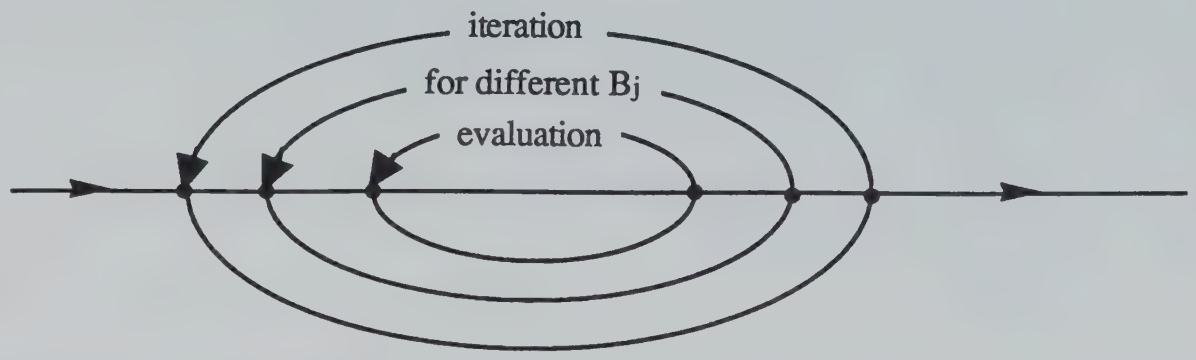


Figure 6.2. Three loops in the radiosity equations.

Let us re-examine the radiosity equations (2.2). With a single processor, there are three nested loops, as shown in Figure 6.2. The inner loop evaluates the formula:

$$\sum_{1 \le i \le N} B_i F_{ji} \ ,$$

the middle one runs for different B_j, and the outer loop performs iteration to gradually approach the accurate solutions. If one of these loops is implemented by a parallel algorithm, the total computation time will be reduced by the number of concurrent tasks running on different processors. Our method divides the middle loop into several components and distributes them in the 64 processors of SPS-2.

Generally, suppose there are n PEs available. n is not necessarily always 64 even in the SPS-2 because very often only half or fewer PEs are available. Thus it is necessary to design a generic algorithm for any n. To distribute the computations, the radiosity equations are partitioned into n groups, each containing $k = \lceil N/n \rceil$ equations. Note that some groups may have fewer than k equations. Let B_1', ..., B_N' contain the radiosities either being initial values E_1, ..., E_N, or obtained by the previous iteration, and B_1, ..., B_N

be the variables to contain new values at the end of the current iteration. At the beginning of an iteration, each group of equations is assigned to a PE by using the **pardo** statement. Thus

$$PE_1 \leftrightarrow (B_1, ..., B_k)$$
$$PE_2 \leftrightarrow (B_{k+1}, ..., B_{2\times k})$$
$$\vdots$$
$$PE_m \leftrightarrow (B_{m\times k+1}, ..., B_{(m+1)\times k})$$
$$\vdots$$
$$PE_n \leftrightarrow (B_{n\times k+1}, ..., B_N)$$

Thus, PE_m is in fact evaluating the following equations:

$$B_j = \frac{1}{1-\rho_j F_{jj}} (E_j + \rho_j \sum_{1\le i\le N, i\ne j} B_i' F_{ji}) \qquad j=m\times k+1, ..., (m+1)\times k \tag{6.1}$$

where normally, $F_{jj} = 0$. After all the PE's finish the above computation, B_j' is replaced by the new value B_j, which will be used in the next iteration.

The parallelism is restricted inside an iteration. From an iteration to the next, the parallel processors have to pause to output the new B_j for the merging operation. After merging B_j, the computations are distributed into different PE's again for the next iteration. Computation from the single processor to the distributed ones requires the system housekeepings that certainly have impact on the parallel performance. This is the most disadvantageous aspect in this parallel scheme.

6.5 More Investigation on Parallelism in Radiosity

We have seen parallelism at different levels: one in calculating separate rows of F_{ij} in parallel, and the other for solving radiosity solutions concurrently. A higher level parallelism indeed exists in radiosity. The following algorithm is invented to accomplish the sophisticated parallel radiosity.

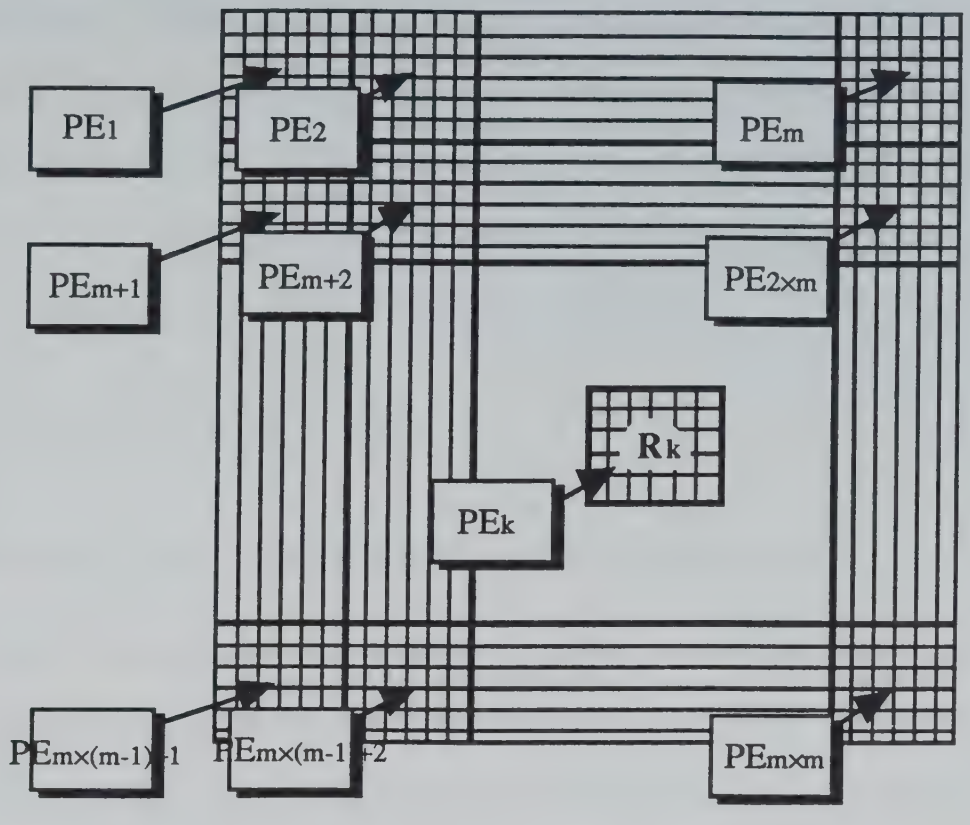


Figure 6.3. Pixel partitioning on the hemi-cube and the associated processors.

Recalling the hemi-cube method in the **FBRM**[1], calculating a row of F_{ij} ($j = 1, ..., N$) takes three steps:

1. sort patches with respect to $\mathcal{P}_i$ ($i = 1, ..., N$);
2. project them in order onto the hemi-cube;
3. accumulate delta form-factors for the visible pixels on the top side.

In order to implement these steps in parallel, the top side of the hemi-cube is partitioned into $m \times m$ rectangular regions, each associated with a processor as shown in Figure 6.3.

[1]Without loss of generalization, only the top side of the hemi-cube is considered.

Basically, a PE is only responsible for accumulation of delta form-factors in its associated region. When the region is fully covered, the PE stops and continues to calculate F_{ij} (j=1,..,N) for a new i no matter whether other PE's have finished their corresponding jobs for the i. Let $F_{ij}(k)$ represent the results from PE_k. When all PEs finish F_{ij} (for some i and all j), $F_{ij}(k)$ are output for the merging operation. All $F_{ij}(k)$ for the same i and the same j are summed up to yield the total F_{ij}[1]:

$$F_{ij} = \sum_{k=1}^{m \times m} F_{ij}(k) \tag{6.2}$$

The scenarios of the parallel algorithm are described in detail as follows.

In order to efficiently implement the algorithm, it is assumed that each PE is able to perform polygon clipping operations by the hardware. For convenience, the whole $m \times m$ PE's are called *FF-chip*. In addition to the *FF*-chip, a general-use PE_0 is used to perform other computations that will become clear in the following discussion. The entire architecture is illustrated in Figure 6.4. To make this parallel system workable, many technical details about the connections between PE_0 and *FF*-chip needed to be considered and added into the scheme. Here we only discuss the ideas on the principles of the architecture. Our main focus is placed on the functionality of *FF*-chip and PE_0, and the data flow between *FF*-chip and PE_0.

PE_0 inputs the database of a scene with attributes like lighting and color properties. The 3-D subdivision is then executed in PE_0 to produce the octree and patches, $\mathcal{P}_1$, ..., $\mathcal{P}_N$. PE_0 chooses $\mathcal{P}_1$ and determines the octant priority with respect to the center of $\mathcal{P}_1$. The transformation is performed on all $\mathcal{P}_i$ (i = 1, ..., N) according to $\mathcal{P}_1$. In the next step, PE_0 sends the transformed $\mathcal{P}_i$ with an octant priority list to all PE's in *FF*-chip so that each PE_k possesses a copy of $\mathcal{P}_1$, ..., $\mathcal{P}_N$ and the octants, O_1, ..., O_l, in a priority order. Then each

---

[1]Precisely, the total F_{ij} should be equal to the sum of F_{ij} for five sides of the hemi-cube.

PE_k starts the hemi-cube process with its own copy of data to calculate partial $F_{1j}(k)$ (j=1,...,N) over on its own region, R_k. The computation of form-factors on R_k is slightly different from the sequential method.

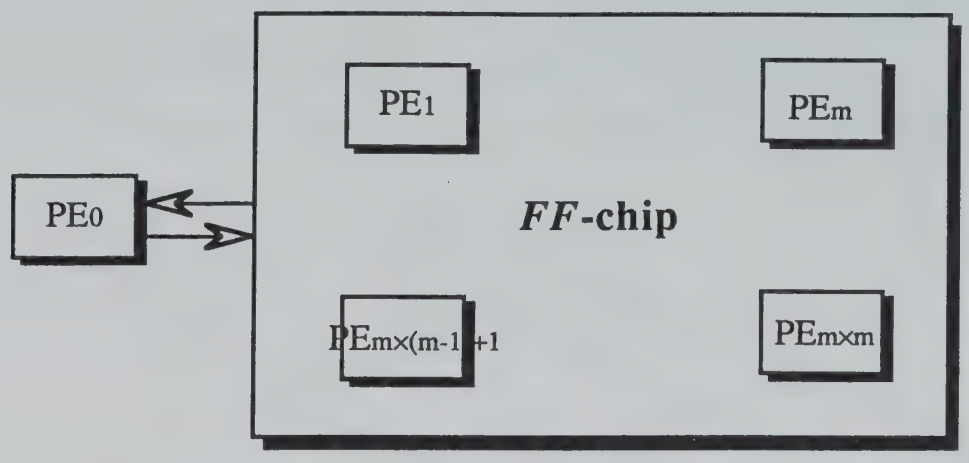


Figure 6.4. The configuration of the parallel system for the form-factor calculation.

As shown in Figure 6.5, a pyramid volume is constructed from the center of the emitting patch (i.e., $\mathcal{P}_1$ at the current stage of process) piercing through the four corners of R_k. Octants are tested against the boundaries of the volume. Any octant lying completely outside this volume is immediately discarded. Let O_1', ..., O_h' be the remaining octants that lie inside or partially inside the volume. The normal hemi-cube painting process described in Section 4.2 is then executed on O_1', ..., O_h'. Note that O_1', ..., O_h' must retain the priority order; that is, O_i' has higher priority than O_j' if $i<j$.

When each PE_k is busy in calculating its own partial $F_{1j}(k)$ concurrently, PE_0 starts processing the next emitting patch, $\mathcal{P}_2$. The sorting operation and the transformation are performed according to the center of $\mathcal{P}_2$. Whenever a PE_k finishes $F_{1j}(k)$, the partial $F_{1j}(k)$ is output to PE_0, and PE_0 passes the new octant ordering and the transformed patches for $\mathcal{P}_2$ to this released PE_k. At this point, PE_0 keeps receiving results $F_{1j}(k)$

returned by the other finished PE_k and merging them. At the same time, each finished PE_k carries on $F_{2j}(k)$ based on the new emitting patch and the new data passed in. When the last PE finishes with $F_{1j}(k)$, PE_0 merges the last results and obtains the total form-factors $F_{1j}=\sum F_{1j}(k)$ for $j = 1, ..., N$. Then PE_0 chooses the third emitting patch, $\mathcal{P}_3$, forms the octant priority, and transforms $\mathcal{P}_i$ ($i = 1, ..., N$). After that, PE_0 is ready to receive the second round-up results, $F_{2j}(k)$ ($j = 1, ..., N$), returned by PE_k.

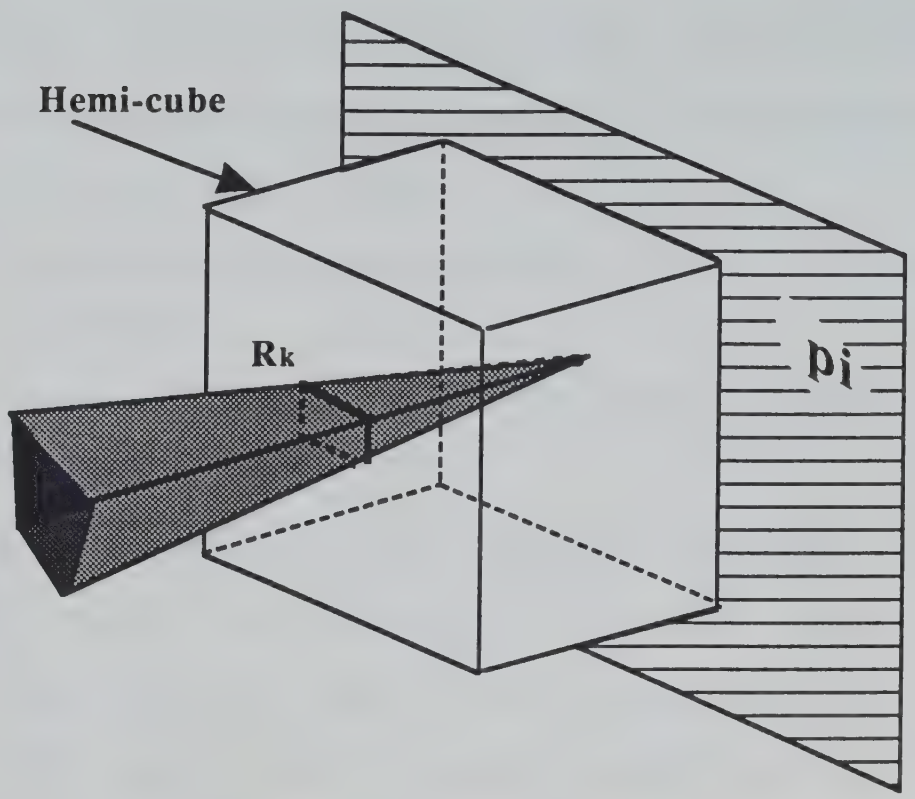


Figure 6.5. The view volume piercing through the small region R_k can be used to determine those octants (outside the volume) that receive no light from p_i and discard them.

The above process repeats until all $\mathcal{P}_1, ..., \mathcal{P}_N$ have been chosen as the emitting patches. At the end of the computation, PE_0 holds the total form-factors, F_{ij} ($i,j = 1, ..., N$), for the top side of the hemi-cube. In order to obtain the complete form-factors, the process has to be repeated again for each side face of the hemi-cube.

When all F_{ij} $(i,j = 1, ..., N)$ are obtained, they are sent back to *FF*-chip for the radiosity solutions. The parallel algorithm stated in the preceding section can be used again with the following modification: the merging operation is done in PE_0. Thus, PE_0 holds the final radiosity $B_1, ..., B_N$ when every PE_k has finished its computation.

To make full use of the *FF*-chip, the rendering step in the radiosity can also be implemented in *FF*-chip in parallel. Before the radiosity, $B_1, ..., B_N$, are sent into *FF*-chip, PE_0 has to complete the pre-processing as follows:

a. Specify a view point, V, and the view volume, based on the application options. V can be treated as the center of the emitting patch in calculating form-factors.

b. Sort the octants with respect to V.

c. Transform the entire modelling database, $\mathcal{P}_1, ..., \mathcal{P}_N$, according to V and the view volume.

d. For each $\mathcal{P}_i$, attach to it an attribute list including the emission, reflectivity, and B_i.

Then PE_0 distributes the data into PE_k, each having a copy of the transformed patches, $\mathcal{P}_1, ..., \mathcal{P}_N$, with their corresponding attributes and radiosity $B_1, ..., B_N$, an ordering of octants, and the view point V. Based on the given data, a similar algorithm for calculating $F_{ij}(k)$ is applied in each PE_k to produce the partial image related to R_k. Note that the algorithm needs to be modified as follows in order to output pixels instead of $F_{ij}(k)$.

In the first step, the top-face of the hemi-cube is replaced by the image plane on which the image is eventually displayed. The region R_k associated with PE_k consists of a raster of pixels instead of delta form-factors. Secondly, when patches are projected onto the image plane (more precisely R_k), the colors at the visible pixels of the projection are calculated and stored in a 2-D array with 24 bits per element. All these pieces of raster

images are merged together in PE_0 to form a complete image that is finally shown on the display device.

What characteristics does the above parallel scheme have? First of all, the simple pipeline between PE_0 and FF-chip reduces the heavy traffic imposed on either PE_0 or PE_k. While PE_k is calculating its partial form-factors, $F_{i1}(k)$, ..., $F_{iN}(k)$, PE_0 is preparing for the next row of F_{ij} by transforming patches and sorting octants based on the newly selected emitting patch. The concurrent computation almost eliminates the cost of transformation and sorting, which are expensive in a sequential method.

Secondly, any PE_k does not have to wait for all the other PE's to finish before proceeding to compute the next row of F_{ij}. Thus, the PE's in *FF*-chip can be used more wisely and efficiently than the parallel scheme described in the beginning of this chapter where some PE's are very busy, while others are in idle states.

Most importantly, each PE in *FF*-chip can calculate a row of $F_{ij}(k)$ very efficiently. There are two reasons for this statement. The region, R_k, is so small that the region will be filled up with 1 very rapidly. Once R_k is full, the rest of octants that have not been processed can be discarded and the computation for the partial $F_{ij}(k)$ (for some i and $j = 1$, ..., N) terminates. Another reason is that the view volume is small. Thus, there are only a small number of octants partially or completely falling in the volume since most of octants are clipped away against the boundaries of this volume. Obviously, the projection operations on a small number of octants will take less time to finish. Of course, the overhead for the clipping operation must be small in order to achieve the desired speed-up. As mentioned before, we assume that each PE has a specialized hardware for fast clipping.

Chapter 7

Implementation and Results

The previous six chapters have given the detailed description, algorithms and the extensive materials of the new radiosity method **FBRM**. Implementation is also regarded as an important part of this thesis in order to demonstrate this method's advantages. We have realized all the features in **FBRM** except the shadow preprocessing. In order to perform an adaptive subdivision, we manually split polygons into subpolygons along shadow boundaries as described in Section 3.4.2. The 3-D subdivision then divides subpolygons into patches using the criteria C_1, C_2 and C_3 (without the shadow test) plus another criterion similar to C_3 that is a supplementary solution to the subdivision around shadows. Several scenes have been modelled and rendered by this method. This chapter will first give the detail of the implementation, and then compare the method with previous ones by analyzing the statistical results with several color images. Advantages and disadvantages of the **FBRM** are also addressed in this chapter.

7.1 Implementation of FBRM

The **FBRM** was implemented and tested on several machines: Sun workstations, *IRIS* and I^2S, as well as *Myrias SPS-2* for the parallel implementation. The software system is called *Myrad*. *Myrad* consists of several modules, each performing one of the four steps: 3-D subdivision, *FF* calculation, radiosity solution, and rendering. The first two modules constitute most of the code of *Myrad*.

7.1.1 Configuration of Hardware and Software

The principal machine used for running *Myrad* is a Sun 3/60. The *I²S* and IRIS are only used to display the color images calculated by *Myrad* running on the Sun 3/60. By using the frame-buffer package designed by Professor Green [Gree87], an image is first stored in a file in the device-independent frame-buffer format. The file is then transferred to either the *I²S* or *IRIS* and converted into the visible color image on their color monitors. The overall organization of the hardware used by *Myrad* is shown in the following diagram.

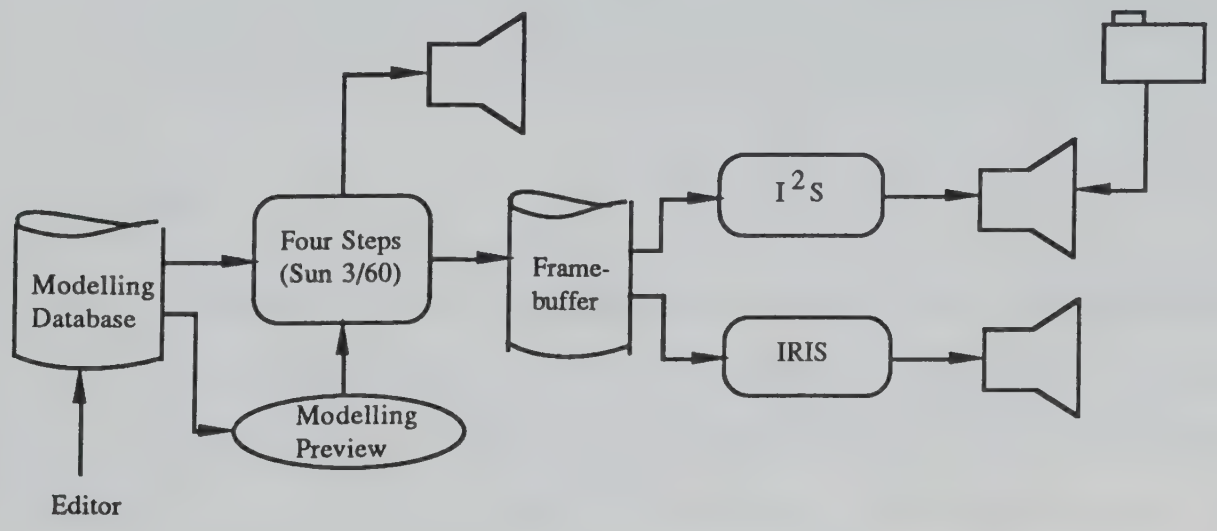


Figure 7.1. The overall organization of the hardware used in *Myrad*.

The system starts with reading in the modelling database describing the scene to be displayed. The modelling database is manually calculated and stored in files. There are two files storing the database: one containing the primitives – polygons, vertices, and attributes, and the other containing the object hierarchy. The primitive file has the following format.

vertBegin
v_1: x_1,y_1,z_1;
v_2: x_2,y_2,z_2;

.
.
.

vertEnd

polyBegin

p_1: v_{11},v_{12},...;
p_2: v_{21},v_{22},...;
.
.
.

polyEnd

attrBegin

a_1: $p\_r_1, p\_g_1, p\_b_1, E\_r_1, E\_g_1, E\_b_1$;
a_2: $p\_r_2, p\_g_2, p\_b_2, E\_r_2, E\_g_2, E\_b_2$;
.
.
.

attrEnd

All vertices are enclosed between **vertBegin** and **vertEnd**. Each vertex starts with a vertex name (a string) followed by x, y, and z coordinates of the vertex. Similarly, polygons are enclosed between **polyBegin** and **polyEnd**, each starting with a polygon name followed by a sequence of vertices (represented by their names) that form the polygon. The content between **attrBegin** and **attrEnd** define a set of attributes for polygons. Each attribute starts similarly with a attribute name followed by six numbers representing the reflectivity and emission in R, G, B, respectively. These attributes will be assigned to each polygon to specify the polygon's material properties that are needed for radiosity solutions. The assignment is done in the object file.

An object file contains a sequence of objects taking one of the following forms.

scene-name
SCENE
n
$o_1, o_2, \ldots\ldots, o_n$;

t_x, t_y, t_z;
s_x, s_y, s_z;
r_x, r_y, r_z;
(for each o_i)

object-name
CLUSTER
n
o_1, o_2, ..., o_n;
t_x, t_y, t_z;
s_x, s_y, s_z;
r_x, r_y, r_z;
(for each o_i)

object-name
PRIMITIVE
n
p_1, p_2, ..., p_n;
a_1, a_2, ..., a_n;

Such a file describes a hierarchic structure of objects in a scene. The object hierarchy, as shown in Figure 7.2, is termed the *object graph.*

There are three type of objects in a modelling database. An object with type *SCENE* represents the whole scene which is the root of the object graph. For each scene, there is only one such object. Objects with the *PRIMITIVE* type are those whose children are polygons, that is, the parents of leaves in the object graph have a type *PRIMITIVE.* Each polygon in a *PRIMITIVE* object is associated with an attribute defined in the primitive-file. The other kind of objects appearing between the root and *PRIMITIVE* objects in the object graph have a type *CLUSTER*, each being a cluster of either other *CLUSTER* objects or *PRIMITIVE* objects.

In the object file, o_1,...o_n point to other objects by their names. For each of these objects, a set of transformation parameters is specified, containing x, y, z components of

each of the basic transformations: translation, scaling, and rotation. These transformation parameters are input and converted into matrix. Then the internal representation of objects is created by traversing the object-graph using a depth-first searching strategy. When a *PRIMITIVE* node is visited, the traversal returns to its parent. Each polygon in the *PRIMITIVE* object just visited is transformed by applying the corresponding transformation matrix and then a copy of the transformed polygon is added into a polygon list belonging to the parent object. When all child objects' polygons are transformed and added into the polygon list, the polygons in the list form the parent object. The parent's parent node (grandparent) is then revisited when returning from that parent node. The above process continues with those polygons in that parent object and other objects that are children of the grandparent. If polygons belonging to these grandparent's children have not been obtained yet, the same procedure is recursively applied to these objects. When we return to the root, the polygon list belonging to the root contains all the polygons that **FBRM** will work with.

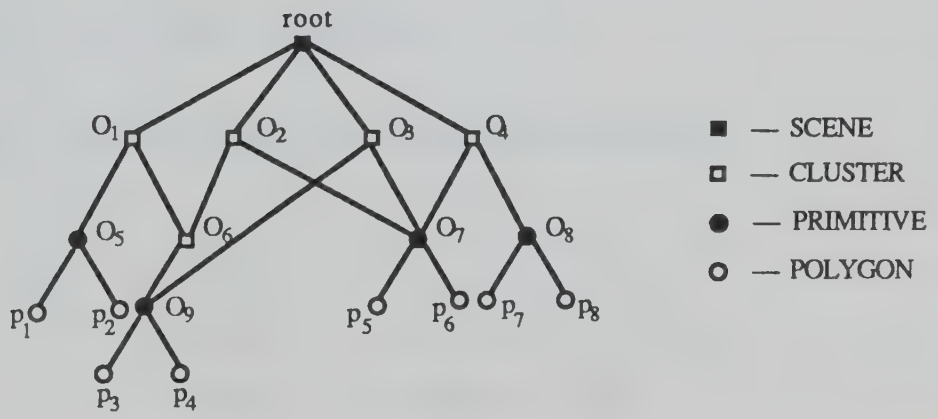


Figure 7.2. An example of object hierarchy.

From the above discussion, the modelling database adopts a string-based scheme for vertices, polygons, attributes and objects. The advantage of this scheme is that it is

easy to manually construct, edit, change, or modify the database. It is easy to merge different databases as well.

The modelling database is read in and sent to the four modules that constitutes the radiosity program, *Myrad*. Before the computation actually starts, the database can be previewed with a program displaying the database in either a frame or solid pseudo image on the color Sun 3/60. The preview facility makes it possible to correct errors such as missing vertices or polygons, cracks, and displacement in the modelling database. The pseudo image is obtained by setting the *B* file (radiosity solutions) randomly and calling the rendering modules with the frame-buffer option off and the display option on. It has been found that this program is very important to assure that *Myrad* works correctly.

After the correct modelling database is read in, the Sun 3/60 executes the four steps of **FBRM** and outputs the 24-bit-per-pixel frame-buffer that contains the final color image. The frame-buffer data is copied to I^2S or *IRIS* where the frame buffer is converted into the image, by **ishow** (part of the *frame-buffer* package), on the color monitor that is capable of producing millions of colors. Several pictures are shown in the color plates.

7.1.2 Four Modules of *Myrad*

As mentioned before, *Myrad* consists of four modules:

Module 1 — performs 3-D subdivision,
Module 2 — calculates form-factors,
Module 3 — solves radiosity equations,
Module 4 — generates the image.

These modules are connected sequentially through the intermediate data structures. Figure 7.3 shows the communication mechanism between the modules.

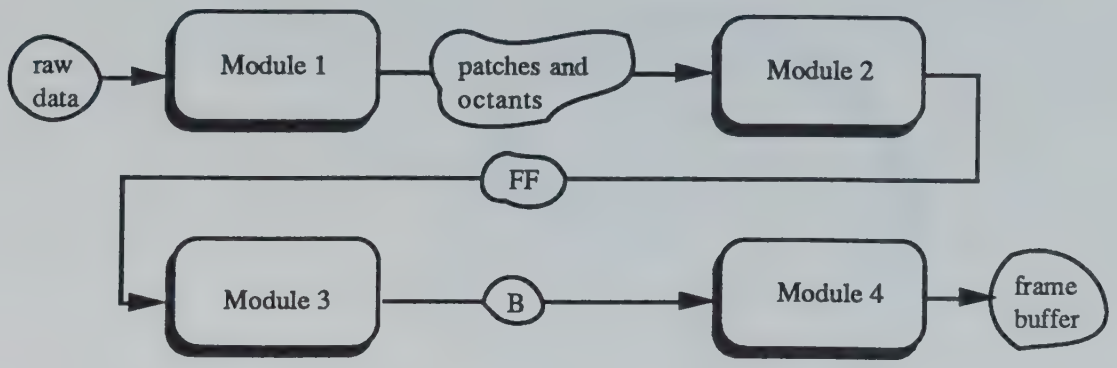


Figure 7.3. The data flow between the four modules in *Myrad.*

In Figure 7.3, *Module 1* inputs the raw data from the modelling database and outputs a set of patches and an octree. Based on the octree, *Module 2* implements the front-to-back hemi-cube algorithm to calculate $N \times N$ form-factors, F_{ij} $(i,j = 1, ..., N)$. The form-factors are input to *Module 3* which solves the N simultaneous equations and produces the solutions, $B_1, ..., B_N$. With inputs, $B_1, ..., B_N$, *Module 4* implements the front-to-back hemi-cube algorithm with a slight modification to generate the image and store it in a frame-buffer.

Although the communication between each pair of modules is performed by the simple file, each module itself has very different complexity in terms of the execution time. The dominant module *Module 2* is the most expensive part, taking up to 93% of the total execution time. The chart in Figure 7.4 illustrates the distribution of computation in *Myrad.*

Usually, it takes hours or even days to render a radiosity image. The expensive computation demands that the intermediate results be saved in files in case of abortion of the execution. Therefore, *Myrad*, is designed to be able to implement the modules independently. And for each module, the resulting data is formatted and stored in files.

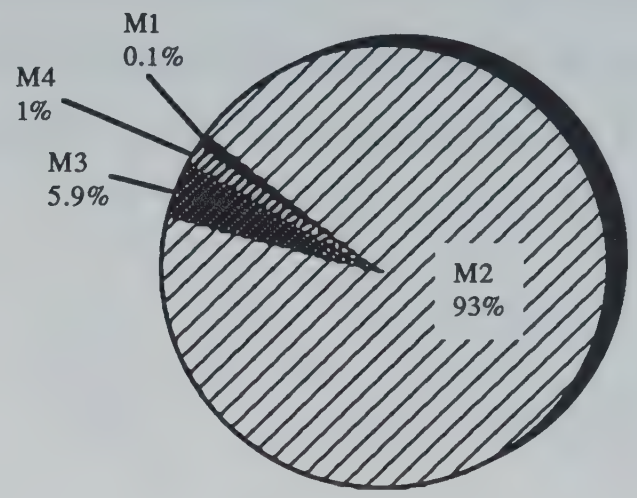


Figure 7.4. Empirical analysis of time complexity distributed in the four modules.

Depending on the complexity of a scene, the whole system can have several configurations. The diagrams in Figure 7.5 shows several examples of configurations for the 4 modules. In fact, any combination of the modules is workable. The connections between modules are performed through files. *Module 1* writes the internal representation of the model including a set of patches and a octree into a binary file. This file is particularly important to the parallel implementation of **FBRM** on Myrias SPS-2 in order to lessen the impact of the bottle-neck of Module 1.

Module 2 has more sophisticated mechanisms to deal with abortion because it takes the most computation time among the four modules. Obviously, when something goes wrong or *Module 2* aborts at the middle of the process, it is unfortunate to lose the data that have been calculating for hours. As a result, an elegant mechanism is specially designed to assure that the data is saved and can be recovered if errors appear.

First, form-factors can be computed selectively based on a set of numbers in pair input to *Module 2*. These numbers represent the rows of F_{ij} that will be calculated by *Module 2*. For example, to calculate only F_{11}, ..., F_{1N}, F_{21}, ..., F_{2N}, F_{31}, ..., F_{3N} and

F_{81}, ..., F_{8N}, the input of numbers can be (1, 3) and (8, 8). Thus to calculate all F_{ij}, the input is $(1, N)$. This consideration allows the system to resume the form-factor computation without repeating computation if *Module 2* stops due to various reasons.

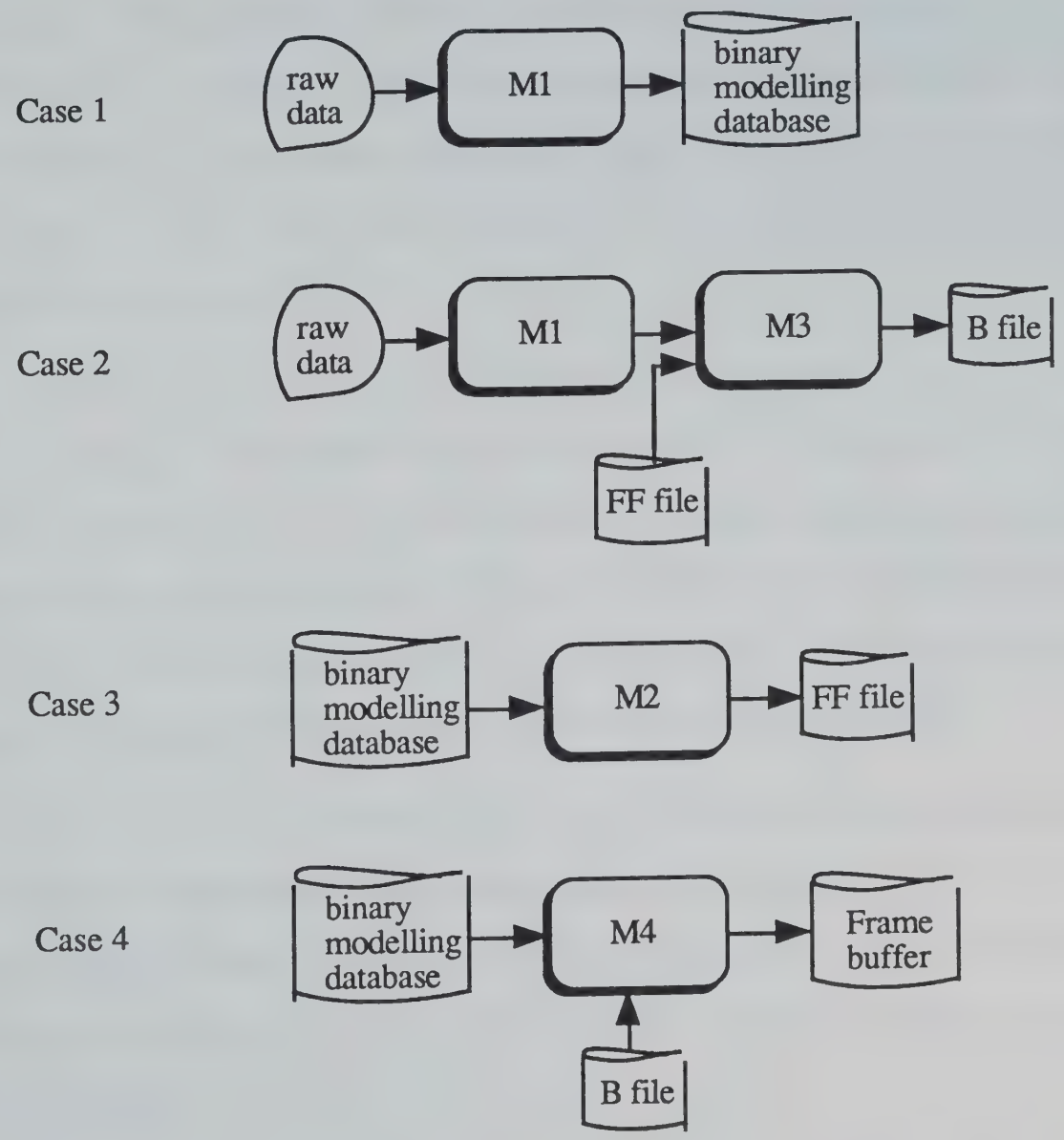


Figure 7.5. Several cases of configuration of the four modules.

The second consideration is that when every row of F_{ij} is finished, the results are flushed to a file called the *FF* file and appended to the row flushed in at the last time. Thus each row of F_{ij} occupies a block in *FF* file. A complete *FF* file consists of N blocks. To

distinguish a row, an **ID** is stored before the block of this row. Since a *FF* file is very huge even in the binary format, a run-length compression is used. This is very efficient for the *FF* file because, on average, 80% of form-factors are zero[1].

Thirdly, a separate program called ***ckFF*** is designed to check validation of the *FF* file off-line. ***CkFF*** checks the *FF* file against two criteria: the number of F_{ij} in a block, and the sum of F_{ij} for a block. A correct block must contain N F_{ij} and the sum of these F_{ij} must be equal to 1[2]. ***CkFF*** is also able to delete a block. Therefore, if one of the criteria is not met for a block, ***ckFF*** deletes this entire block and then *Module 2* is run to recalculate the block and append it to the *FF* file.

Module 3 simply outputs an array of radiosity solutions, B_1, ..., B_N into the *B* file. It is useful to save the *B* file for later use. Particularly, it can be reused to constantly display the same image with only the viewpoint moving and the scene's geometry unchanged. Saving the *B* file and the *FF* file will also help animation generation when only the viewpoint or the material properties like colors are changing. Note that the *FF* file does not have to be recalculated if only colors or the viewpoint has been changed.

The output of the last module, *Module 4*, is a compressed binary file containing the device-independent frame-buffer that can be re-displayed on any color monitor with 24-bits per pixel. The objectives to save images into files are to allow images to be portable, to review images as many times as we want, and to make playback of simple sequence of animation possible.

So far, we have explored the external data structures and the data flows among the four modules. The internal data structures, and the details of implementation within each

---

[1]Since 80% of form-factors are 0, a sparse matrix method can be used for solving the radiosity system of linear equations efficiently. This thesis is, however, mainly concerned with form-factor calculation. We expect to include the method in our future implementation of **FBRM**.

[2] Because of computer round-off and discretization errors, it is found that 0.999 instead of 1 would be the best result.

module will not be described because of their complexity. The formal description of these algorithms for Module 1 and *Module 2* have been presented in Chapter 3 and Chapter 4. *Module 3* is implemented based on Gauss-Seidel algorithm. Thus there is no need to describe its implementation here. Since the rendering part, *Module 4*, borrows an algorithm similar to the front-to-back used in *Module 2*, its technical detail is also omitted.

7.2 Comparison and Results

As shown in the previous chapters, this thesis is dedicated to solving radiosity problems, mainly from three aspects: speed, anti-aliasing, and parallelism. The fundamental algorithms described through Chapters 3 to 4 have enhanced the hemi-cube method by ordering octants and painting patches from front to back. Based on these algorithms, an anti-aliasing flavor is also added to improve quality of images. In addition, this thesis also explores parallelism in radiosity and gives an in-depth description of parallel implementation. Although the detailed methods or algorithms related to the three aspects have been stated, a full analysis of the results produced by these methods is necessary to understand their advantages and disadvantages.

Five scenes with different complexity have been modelled and rendered by *Myrad*. The images for these scenes are shown in the color plates 1 to 5. Also the implementation results are shown in Tables 3 to 8. To highlight advantages of the **FBRM**, we compare the implementation results from four aspects: the front-to-back algorithm and the z-buffer method, crowded scenes and less crowded scenes, aliasing and anti-aliasing methods, and sequential and parallel algorithms.

7.2.1 Front-to-back vs z-buffer

Basically, the new method, **FBRM**, is different from the traditional hemi-cube method in two aspects. First, the new algorithm implements the polygon subdivision in the 3-dimensional space and produces ordered patches implied in the octree, while the traditional

hemi-cube method subdivides polygons into unordered patches along with the surfaces of polygons. Secondly, in the new algorithm, the form-factor calculation is accomplished by a front-to-back algorithm, while the traditional hemi-cube method uses the z-buffer that sorts patches based on the z values. As a result, comparison between the front-to-back and the z-buffer can give a good evaluation on the new radiosity method.

The front-to-back has a major advantage over the z-buffer. The complexity for calculating a row of F_{ij} is dependent on the position of the emitting patch and the surrounding patches. If the emitting patch only sees a few surrounding patches, computing the corresponding row of F_{ij} will terminate quickly. However, the z-buffer has a constant time for calculating any row of F_{ij}. This is because the former deals with ordered patches, while the latter handles a set of unordered patches.

Of course, sorting patches requires additional overhead. The question is: will the front-to-back be slower than z-buffer due to the sorting overhead? The following analysis answers this question.

By careful examination, the front-to-back has an overhead consisting of indexing the *basic octant priority* table (***BOPT***), and returning the parent cube at each step in traversing the linear octree (***FOL***). As far as the overhead of determining octant priority is concerned, the z-buffer seems to be better than the front-to-back. But during the process of form-factor calculation, the z-buffer has the extra computation for z values which the front-to-back does not have. In fact, the z-buffer and the front-to-back have the pretty much similar scheme except the operations at each pixel. The z-buffer has four operations at each pixel, while the front-to-back without anti-aliasing has three.

z-buffer	**front-to-back**
comparison operations	comparison operations
computing z values	updating 1-bit mask
updating z buffer	updating variable ***FF***[][].
updating item buffer	

The extra computation in z-buffer does not really affect the speed if the above computation is performed only once. But computing whole form-factors requires execution of the above computation millions of times. The final outcomes indeed make a lot of difference. The overhead with the z-buffer may cancel off the overhead imposed by the octree sorting in front-to-back. In particular, when the size of the octree is smaller than the number of pixels of each image or each face of the hemi-cube, the front-to-back is in fact faster than the z-buffer. Suppose **FBRM** produces an octree with 4-level depth which is good enough for a fine subdivision. In the worst case, the size or the number of nodes of the octree is $8+8^2+8^3+8^4 = 4680$. That means that the time for octree sorting requires 4680 operations. On the other hand, if the resolution of the hemi-cube is 100×100, the number of the extra operations that the z-buffer has to perform is 10000. Usually, to reduce aliasing, the resolution can be 300×300 = 90000. If the number of patches, N, is equal to 2000, the difference of the total operations between the z-buffer and the front-to-back for calculating complete $N \times N$ form-factors is

$$90000 \times 2000 - 4680 \times 2000 = 170640000$$

Suppose that each operation takes 100*ns*; therefore, the z-buffer will spend 4.74 hours more than the front-to-back. Based on the above observation, Figure 7.6 gives a simple comparison between the two methods. The comparison uses two functions: $T = R^2 \times N \times \tau$ for the z buffer, and $T = (8+8^2+...+8^D) \times N \times \tau$ for the front-to-back, where T represents the time, R is the resolution of the hemi-cube, N is the number of patches, D is the depth of the octree, and τ is the time spent by each operation. (τ is assumed to be equal to 100*ns*.)

Traversing the octree is performed only once for all five faces of the hemi-cube, while the z buffer has to be gone over five times for different faces of the hemi-cube. This implies that the overhead in traversing the octree is indeed small compared with that in the z buffer.

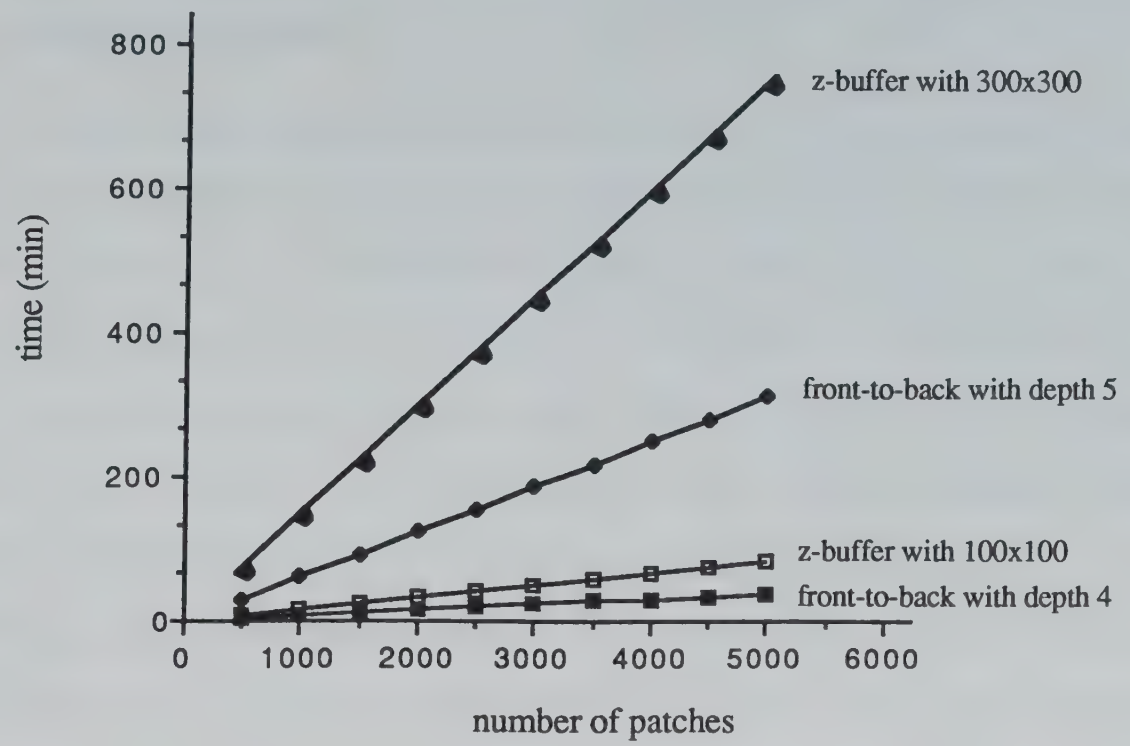


Figure 7.6. Comparison between z-buffer and front-to-back.

We derive the above analysis based on an assumption that the patches are not overlapped at all. If many patches happen to overlap with others, more than one operations can be performed at one single pixel. Therefore, the z-buffer will spend much more time than the front-to-back for form-factor calculation.

7.2.2 Crowded Scenes vs Less Crowded Scenes

To observe the pros and cons of the front-to-back and the z-buffer from a higher level, a new measurement for comparison is established — the number of octants that have been actually processed for a row of form-factors.

The z-buffer does not involve the concept of *octant*. In order to compare the front-to-back with the z-buffer in terms of octants, we can assume that each patch dealt with by the z-buffer is enclosed in a virtual octant. Those octants can be the same as those used in the front-to-back. In other words, we are in fact comparing two methods, one with 3-D

subdivision and the front-to-back, and the other with the same 3-D subdivision but the z-buffer without sorting octants.

From the statistical results shown in Tables 3 to 7, the front-to-back only needs to process less than 50% (on average) of all octants in the environment and discard the rest. But the z-buffer has to go through all the octants in order to determine visibility. The reason why the front-to-back can save almost 50% time is that the front-to-back does not have to process octants further once the mask table is full of 1's. This is because the patches in the discarded octants must be hidden from the patches processed. The time complexity of the front-to-back is quite different from one emitting patch to another, depending on the geometric relation between the emitting patch and the other patches. On the other hand, the z-buffer has no knowledge of which patch occludes the other until all patches have been processed. The complexity is a constant to any emitting patch, or any row of F_{ij}.

In most situations, an emitting patch can only "see" part of the environment, because the emitting patch normally appears inside the environment. This implies that the mask table can be filled up very fast and the painting process for the emitting patch will terminate very fast. From the algorithm described in Section 4.3.4, this is reflected by the number of the loops running through ***OPL***. If the mask table is full very quickly, only a small number of octants in ***OPL*** will be processed, the rest being discarded. Obviously, the smaller the number of octants processed, the faster the algorithm.

From the analysis above, the front-to-back can do even a better job when the environment is crowded with objects. The more crowded an environment, the fewer objects an emitting patch can see and the faster the process terminates.

The following example reveals the advantages of the front-to-back more clearly. Compare color plate 2 with color plate 4 and their corresponding results in Tables 4 and 6.

The scene in Plate 2 is occupied by loosely distributed objects with about 1059 patches and 481 octants. From Table 2, the average number of octants processed is 292 per row of form-factors per face of hemi-cube. The total time for all form-factors is 430 min. Plate 4 shows an environment with crowded cubes in the space. The number of patches in the scene is equal to 1170 and the number of octants is 932. On average, however, only 170 octants are actually processed for a row on a face. The total time for obtaining the complete set of form-factors is equal to 234 min. This example shows that more crowded scenes need less time for form-factor calculation than less crowded ones.

However, carefully checking Tables 3 to 7, it is found that the z-buffer is not only much slower than the front-to-back, but also will spend more computation time for a crowded scene than a less crowded environment. In general, the computational complexity of the z-buffer increases dramatically with the increase of the number of objects, especially in the case of large objects.

The graph in Figure 7.7 gives a clear picture of comparison between crowded scenes and less crowded scenes by using the front-to-back and the z-buffer. Note that the graph drawn in Figure 7.7 is based on the statistic results for images 1 to 5. They may not accurately reflect the behavior of the **FBRM**, but they do characterize the advantages of the new method.

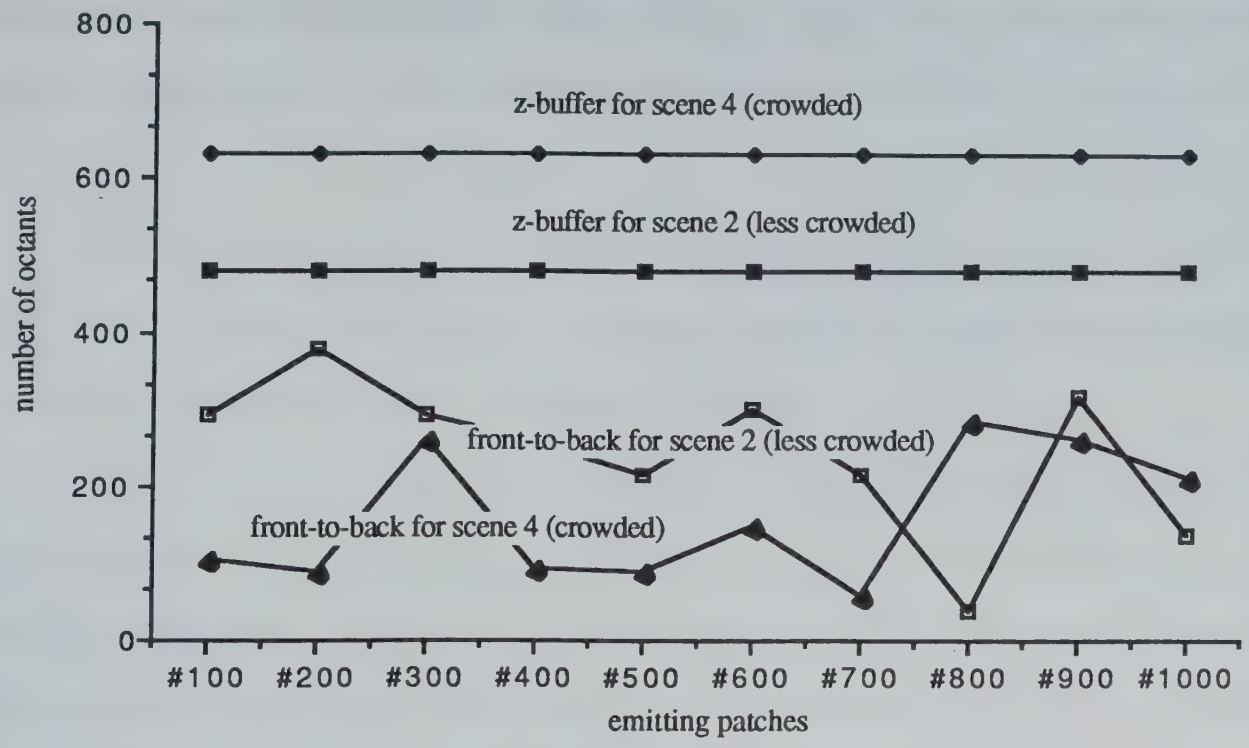


Figure 7.7. Comparison between crowded and less crowded scenes. (the horizontal axis represents the patch identies).

7.2.3 Anti-aliasing vs Aliasing

Since the anti-aliasing algorithm has used a fraction-mask table for visibility computation, the overhead due to the fraction-mask table will have a serious impact on the new method in terms of speed. It may be slower than the z-buffer in some situations, but it definitely gains higher quality of images as the trade-off. Nevertheless, there are many cases, especially in a crowded environment, where the front-to-back with the anti-aliasing is still better than the z-buffer. This is because in most situations, the actual number of octants processed is less than half of the total number of octants; that is, the front-to-back may be twice as fast as the z-buffer. The speed-up can cancel off the overhead produced by the fraction mask operations.

The last rows in Tables 3 to 6 actually show the results that are produced by using the anti-aliasing **FBRM** to render the scenes shown in Plates 1 to 4. The second rows in

Tables 3, 4, 6 and 7 contain the results of using the basic **FBRM** to generate these scenes. Obviously, the anti-aliasing **FBRM** has sacrificed in speed, but it does not lose much speed, particularly in the crowded environments like Scene 4. Overall, it is still faster than the z-buffer.

From image quality point of view, the anti-aliasing **FBRM** achieves better results than either the basic **FBRM** or the traditional hemi-cube method. To support this statement, let us analyze the aliasing problem on the image in Figure 5.1a.

There is only one light source[1] in Figure 5.1. The illumination showing on the object surfaces is contributed mainly by this light source. With this observation, it is evident that the intensity on the surface straight under the light is the highest. The other surfaces should comply to the rule: the closer to the highlighted surface they are, the higher their intensity. Starting from the highlighted surface, therefore, intensity along the surfaces should change gradually from high to low. The image in Figure 5.1a, however, does not follow this rule, showing irregular patterns of intensity.

The aliasing in calculating form-factors is the cause of the irregular patterns. Consider Figure 7.8 in which three close patches $\mathcal{P}_1$, $\mathcal{P}_2$ and $\mathcal{P}_3$ are projected onto the hemi-cube in order to determine the form-factors F_{l1}, F_{l2} and F_{l3} from the light to $\mathcal{P}_1$, $\mathcal{P}_2$ and $\mathcal{P}_3$ where l is the index of the light source. Suppose that $\mathcal{P}_1$ is projected first, $\mathcal{P}_3$ second and $\mathcal{P}_2$ at last. All the pixels on the boundary between $\mathcal{P}_1$ and $\mathcal{P}_2$ that are partially covered are considered as part of $\mathcal{P}_1$, and F_{l1} then gets more delta form-factors than it should. Similarly, F_{l3} will have more delta form-factors. However, it is a different story about $\mathcal{P}_2$. Since the boundaries of $\mathcal{P}_2$ that share with $\mathcal{P}_1$ and $\mathcal{P}_3$ have been used by $\mathcal{P}_1$ and $\mathcal{P}_3$ (i.e., these places are set by 1's in the mask table), F_{l2} will get fewer delta form-factors than it should. Recall that a form-factor represents the fraction of energy that leaves the emitting

---

[1] The light does not show on the images in Figure 5.1. In addition, to show the illumination mainly from the light source, there is no wall surrounding the environment, assuming that the walls are at the infinite location.

patch and lands on the receiving patch. As a result, $\mathcal{P}_1$ and $\mathcal{P}_3$ will receive more energy than $\mathcal{P}_2$[1], and thus turn out to be brighter than $\mathcal{P}_2$. Thus the final image does not show smooth change of intensity across patches.

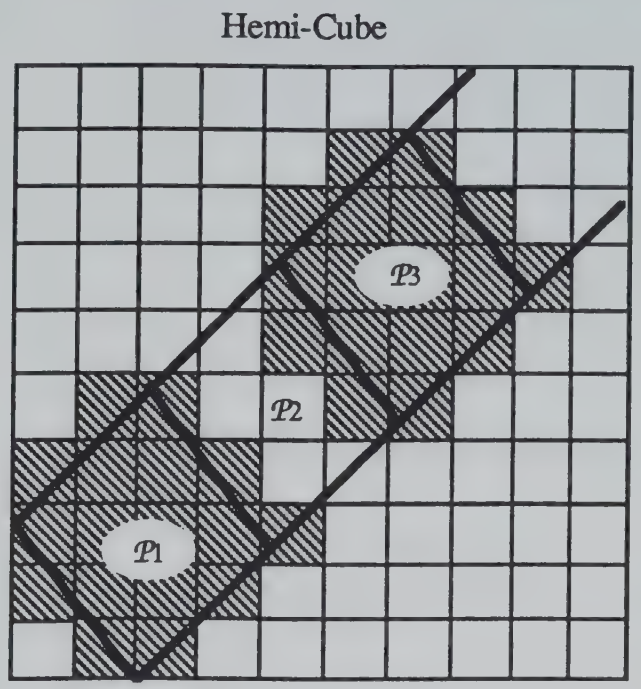


Figure 7.8. Miscalculation of delta form-factors because of aliasing.

On the other hand, the image in Figure 5.1b has symmetric and regular patterns of intensity of patches. This is because the pixels on the boundaries between $\mathcal{P}_1$ and $\mathcal{P}_2$ are split into two parts, one for $\mathcal{P}_1$ and the other for $\mathcal{P}_2$. By using coverage weights applied to delta form-factors, F_{l1} and F_{l2} are determined properly and the intensity for $\mathcal{P}_1$ and $\mathcal{P}_2$ is displayed correctly. Similarly, $\mathcal{P}_3$ is shown by correct intensity.

---

[1] Assume that their sizes are the same and the angles between surface normals and the direction to the light are almost the same.

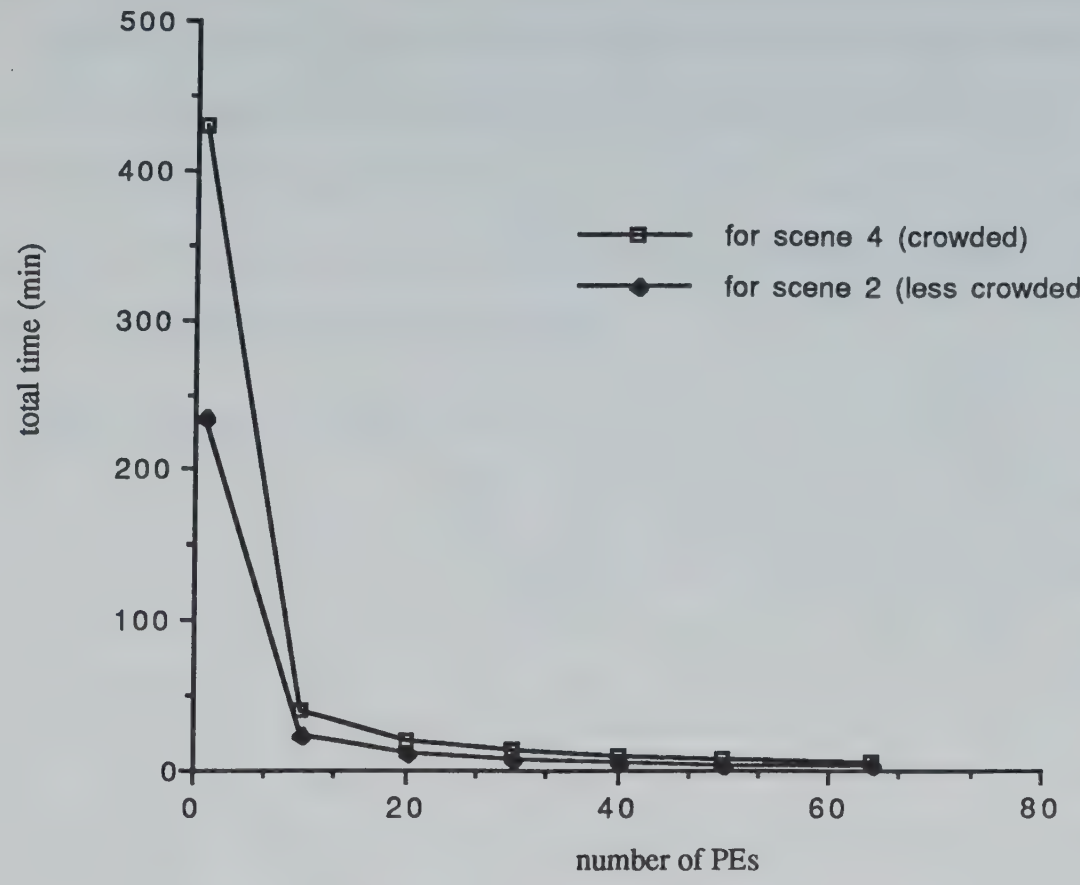


Figure 7.9. Comparison between sequential and parallel implementation of *Myrad*.

7.2.4 Sequential vs Parallel

The speed-up using parallel algorithms is obvious. Table 8 gives some results for Scene 2 and Scene 4 by implementing *Module 2* by sequential and parallel approaches. Figure 7.9 visually shows the results in Table 8. From the graph, using 64 PEs, the parallel algorithm only requires 7 minutes of CPU time for calculating form-factors with anti-aliasing for Scene 2, which needs 430 minutes in the sequential machine. The speed-up is almost linearly proportional to the number of PEs.

The linear speed-up is achieved because of two factors. First, the system time is minimum since there is no cost for merging data except for writing a row of F_{ij} into a file. Secondly, there is little bottleneck since the 3-D subdivision is performed in the front-end processor and the results are written into a file.

We have compared the new method with traditional ones from several points of view by showing actual implementation results and pictures. The new method **FBRM** certainly has many advantages and achieves a better performance in terms of speed and image quality. In the next chapter, the advantages in the **FBRM** are summarized and the future work is pointed out to conclude the thesis.

Table 3 Statistical Results for Scene 1

	# of patches	Time for 3-sub (sec.)	Time for form-factor (min.)	Render time (min.)	Max # of octants	Level of octree	Octants per face
Z Buffer	1098	16	435	5.8	872	5	872
FBRM (no anti-aliasing)	1098	16	290	4.9	872	5	499
FBRM (anti-aliasing)	1098	16	319	5.2	872	5	539

Table 4 Statistical Results for Scene 2

	# of patches	Time for 3-sub (sec.)	Time for form-factor (min.)	Render time (min.)	Max # of octants	Level of octree	Octants per face
Z Buffer	1059	12	496	5	481	4	481
FBRM (no anti-aliasing)	1059	12	354	4.8	481	4	230
FBRM (anti-aliasing)	1059	12	430	5.3	481	4	292

Table 5 Statistical Results for Scene 3*

	# of patches	Time for 3-sub (sec.)	Time for form-factor (min.)	Render time (min.)	Max # of octants	Level of octree	Octants per face
Z Buffer	2329	19	843	9.2	816	5	816
FBRM (anti-aliasing)	2329	19	575	7.7	816	5	267

Table 6 Statistical Results for Scene 4

	# of patches	Time for 3-sub (sec.)	Time for form-factor (min.)	Render time (min.)	Max # of octants	Level of octree	Octants per face
Z Buffer	1170	15	526	6.2	932	4	932
FBRM (no anti-aliasing)	1170	15	194	4.6	932	4	125
FBRM (anti-aliasing)	1170	15	234	4.9	932	4	170

Table 7 Statistical Results for Scene 5**

	# of patches	Time for 3-sub (sec.)	Time for form-factor (min.)	Render time (min.)	Max # of octants	Level of octree	Octants per face
Z Buffer	441	8	100	2	241	5	241
FBRM (no anti-aliasing)	441	8	84	2	241	5	184

* The no-anti-aliasing version of **FBRM** was not run with the data for Scene 3.

** The anti-aliasing version of **FBRM** was not executed with the data for Scene 5.

Table 8 Performance on Myrias SPS-2 Computer

No. of PEs	Scene 2 (min.)	Scene 4 (min.)
1	430	234
10	41.4	23.3
20	21.56	11.66
30	14.93	7.86
40	10.73	5.9
50	8.43	4.72
64	6.98	3.96

Color Plate 1 The image with anti-aliasing in both form-factor calculation and rendering for Scene 1.

Color Plate 2 The image with anti-aliasing in both form-factor calculation and rendering for Scene 2.

Color Plate 3 The image with anti-aliasing in both form-factor calculation and rendering for Scene 3.

Color Plate 4 The image with anti-aliasing in both form-factor calculation and rendering for Scene 4.

Color Plate 5 The image with aliasing in both form-factor calculation and rendering for Scene 5 using the traditional hemi-cube.

Chapter 8

Conclusions and Future Work

The previous seven chapters have basically reviewed the image rendering technique of radiosity from various aspects: fundamental theory, traditional methods, alternative methods, anti-aliasing problems, parallel algorithms, and implementation. The discussions focus on developing the new method, **FBRM** (**F**ront-to-**B**ack based **R**adiosity **M**ethod). Based on the new scheme, a lot of work can be done to improve radiosity methods further by applying it to other methods such as the progressive refinement method, or implementing it on an existing parallel architecture. This chapter will give a full summary of this thesis and then reveal some aspects of further work on **FBRM**.

8.1 Summary

The work that has been done in the thesis includes six aspects:

- Theoretical and fundamental,
- Review of previous methods,
- Basis of **FBRM**,
- Anti-aliasing problems,
- Parallel algorithms,
- Implementation and result analysis.

Any radiosity method basically employs the principles of thermal radiation heat transfer in thermal engineering. According to energy conservation, radiosities or intensities

reflected from surfaces in an enclosed environment are related by the following famous equations:

$$B_j = E_j + \rho_j \sum_{i=1}^{N} B_i F_{ji} \qquad (j = 1,...,N).$$

The form-factor, F_{ji}, is decided by the geometric relation between two patches j and i. The previous radiosity methods attempted to find fast ways for obtaining the form-factors. Among these methods, the hemi-cube method is more popular. The new method, **FBRM**, is based on this hemi-cube method to improve the radiosity method's performance by choosing the front-to-back hidden surface removal algorithm.

Basically, there are two steps in **FBRM**. The first step is the 3-D subdivision of polygons into smaller patches in order to obtain more accurate radiosity. To subdivide polygons, the octree technique is used to clip polygons against the boundaries of octants. The octant priority is the important concept introduced in this thesis. With the octant priority list, the hemi-cube method can be modified to execute a front-to-back algorithm for form-factor calculation. Thus, the second step is to use the front-to-back hidden surface algorithm to compute form-factors, a row at a time, by painting receiving patches on the hemi-cube centered at the emitting patch.

The 3-D polygon subdivision also achieves adaptively extensive subdivision around shadow boundaries and object corners. The shadow information is obtained at a preprocessing step in which a shadow algorithm identifies the lit subpolygons. The shadow information then guides the 3-D subdivision to perform extensive meshing of polygons adaptively. The task is accomplished by specifying the criterion vectors for subdivision termination. The criterion vector is one of the concepts introduced in this thesis. It has the potential in achieving optimal adaptive subdivision.

The major improvement of this new method lies in the second step. Painting patches from the nearest to the farthest onto the hemi-cube has several advantages.

The first and most important advantage is that it is not necessary to paint all the receiving patches to obtain a row of form-factors. In the most cases, more than 80% form-factors are zero. This implies these zero form-factors must be those between patches that cannot "see" each other; in other words, this pair of patches are occluded by other patches. When patches are painted from front to back, the occluded patches do not have to be actually painted onto the hemi-cube. The form-factor for these patches are assumed to be zero. To determine which are occluded, a 1-bit mask table is added to assist the fast form-factor computation. More technically, when the mask table is filled, which means the remaining un-painted patches must not be seen by the emitting patch, computation for this row of form-factors is terminated in the middle.

The second improvement is elimination of two large buffers: z-buffer and item-buffer. This is because the front-to-back algorithm determines the visibility of patches immediately after the patches are painted onto the hemi-cube. (The z-buffer algorithm, however, has to use the z-buffer to determine the visibility and temporarily store the patch currently being painted in the item buffer.) The delta form-factor for the patch being painted is obtained on the fly without waiting for other patches to finish painting. Therefore, there is no need to compare z values and store patches for computing delta form-factors at a later time.

The third advantage is to make it possible to add anti-aliasing into the hemi-cube based form-factor calculation with minor cost. The aliasing problem is very notable in calculating form-factors. It may cause the inaccuracy of form-factors which can have serious negative impact on the final image. Traditionally, the hemi-cube method ignores the problem or increases the resolution of the five faces of the hemi-cube to reduce the aliasing problem. The failure of adding anti-aliasing is due to the z-buffer that is unable to acquire information of the background or other patches behind the current one. However, the front-to-back works with a sequence of sorted patches and knows exactly which

patches lie behind. Thus it is able to do anti-aliasing enhancement by refining the delta form-factors according to the surrounding patches.

In addition to the basic ideas and principles of the new radiosity method, this thesis also includes a detailed implementation on both sequential and parallel system. The sequential implementation is divided into three stages. The initial stage used the traditional hemi-cube with the z-buffer to experiment with the radiosity methods. A couple of simple pictures were generated. In the second stage, the basic **FBRM** without anti-aliasing was implemented. Several color pictures from simple to complex were produced with valuable statistical results that are important to analysis of the new methods. In particular, these results show that the more crowded the environment, the better the method. The third stage of the sequential implementation added the anti-aliasing technique in the basic **FBRM**. The anti-aliasing **FBRM** certainly improves quality of radiosity images, but sacrifice some speed in return. Although the anti-aliasing **FBRM** is slower than the basic **FBRM** because of the use of the fraction mask table, in some cases it is still better than the traditional ones in terms of both quality and efficiency, especially in a crowded environment.

The parallel implementation of **FBRM** is only an experiment to test how the radiosity method fits in the parallel architecture. Although simple, the parallel implementation significantly demonstrates the potentials of radiosity methods in a distributed or multiple processing environment. This thesis only implements the **FBRM** at two levels of parallelism: one for calculating rows of form-factors concurrently, and the other for solving the simultaneous radiosity equations in parallel. In addition, this thesis also proposes a more sophisticated algorithm in a lower level by taking advantage of front-to-back. Based on this proposal, it is possible to develop a practical VLSI algorithm under today's VLSI technology. Accomplishment of this VLSI radiosity algorithm is of great value to the animation industry or scientific image research.

8.2 Future Work

Our new approach to the radiosity area is only a primitive step toward the goal of real-time realistic image generation. Many problems still remain unsolved and further improvements need to be done. Based on the **FBRM**, the following suggestions may become good approaches to improving radiosity.

8.2.1 Applying FBRM to Progressive Refinement

As indicated before, **FBRM** is only an alternative approach to tackle the radiosity method by substituting the z-buffer in the hemi-cube method with the front-to-back. **FBRM** can, therefore, be used in any other radiosity method where the hemi-cube is used. The progressive refinement approach improves the hemi-cube by calculating form-factor on the fly. The **FBRM** can be used to replace the hemi-cube method in the progressive refinement in order to increase the performance further. The following will give some ideas of how to apply the **FBRM** to the progressive approach.

The polygon subdivision is accomplished in the same way as the 3-D subdivision so that octant priority lists can be found as the polygons are clipped into patches. The progressive refinement is then applied using the front-to-back algorithm (with anti-aliasing or without it) to calculate the form-factors. Starting with the patch with highest intensity, like the light sources, the row of form-factors from the patch (emitting patch) to the rest of patches (receiving patches) are calculated by painting the receiving patches onto the hemi-cube from the higher priority patches to the lower ones until the mask table is full. Once the row of form-factors are finished, the radiosity for those receiving patches are determined according to the newly obtained form-factors. The image with the partial radiosity is then displayed instantly. If the image is already sufficient, the whole process can be terminated. Otherwise, the patch with the highest intensity among the receiving

patches is chosen to be the emitting patch. The above process continues with the chosen patch.

The combination of the progressive approach and the front-to-back will significantly speed up the radiosity image generation. The progressive method makes improvement on efficiency by reducing the total number of rows of form-factors to be calculated, while the **FBRM** increases the performance by speeding up the computation for a row of form-factors and avoiding computing the form-factors that are actually zero. Therefore, the total increase of the performance will be the product of both performances. Suppose that each of the progressive method and the **FBRM** needs 50% of the total time that is required by the traditional radiosity method. Thus the combination of two methods will spend 25% of the time required traditionally to obtain a radiosity image. The saving is significant.

8.2.2 Effective Region

An effective region is a 3-dimensional sphere containing an emitting patch located at the center and having a radius equal to the maximum distance at which the light can travel starting from the emitting patch.

It is observed that a less-reflective or less-emitted object contributes light only to a local area. In other words, any other object, say object B, far away from the object, say object A, under consideration receives less radiant energy leaving A. In this case, it can be assumed that the fraction of the energy leaving A and reaching B is approximately zero. This implies that the calculation for the form-factor from A to B can be eliminated. The effective region defines a scheme to eliminate the processing of *far-away* patches. Any patch or octant that lies outside the spherical region is discarded without further processing, and the form-factor from the emitting patch to it is assumed to be zero. The question is how to determine the radius of the sphere.

The size of the effective region is dependent on the material properties and the size of the emitting patch. For example, a light source or a polished surface may have a larger sphere than a small rough object. Forming a set of rules to determine the size of sphere requires sophisticated research and experimental observations.

Note that the rules for determining region sizes can be simplified if the progressive refinement approach is added to the **FBRM**. Since the radiosity of the emitting patch is obtained instantly and dynamically in the progressive method, the distance at which the emitted light can travel should be determined easily based on the value of the radiosity. As a result, the method will turn out to be a very efficient radiosity method in future.

Bibliography

[Am84] Amanatides, J., "Ray Tracing with Cones", *Proc. of SIGGRAPH 84*, Vol. 18, No. 3, July 1984, pp. 129-135.

[AtWe78] Atherton, P.R., Weiler, K. and Greenberg, D.P., "Polygon Shadow Generation", *Proc. of SIGGRAPH 78*, pp. 275-281.

[BaRu89] Baum, D.R., Rushmerier, H.E. and Winget, J.M., "Improving Radiosity Solutions through the Use of Analytically Determined Form-Factors", *Proc. of SIGGRAPH' 89*, pp. 325-334.

[Bl78] Blinn, J. F., "A Scan Line Algorithm for Displaying Parametrically Defined Surfaces", *Proc. of 5th Conference on Computer Graphics and Interactive Techniques* , 1978.

[BrBa84] Brotman, L.S. and Badler, N.L., "Generating Soft Shadows with a Depth Buffer Algorithm", *IEEE CG & A*, Vol. 4, No. 10, October 1984, 5-12.

[Ca74] Catmull, E. E. "A Subdivision Algorithm for Computer Display of Curved Surfaces", *Ph.D. Thesis*, University of Utah, Salt Lake City, Utah, Dec. 1974.

[Ca75] Catmull, E. E., "Computer Display of Curved Surfaces", *Proc. IEEE Conf. on Computer Graphics, Pattern Recognition and Data Structures*, Los Angeles (May 1975)

[Ca80] Carpenter, Loren C., "Computer Rendering of Fractal Curves and Surfaces", *suppl. to Proc. of SIGGRAPH 80*, August 1980, pp. 1-8.

[Carp84] Carpenter, L, "The A-buffer, an Antialiased Hidden Surface Method", *Computer Graphics*, Vol. 18, No. 3, July 1984, pp. 103-108.

[Catm78] Catmull, E.C., "A Hidden-Surface Algorithm with Anti-Aliasing", *Computer Graphics*, Vol. 12, No. 3, August 1978, pp. 6-11.

[CoCh88] Cohen, M.F., Chen S.E., Wallace, J.R., Greenberg, D.P., "A Progressive Refinement Approach to Fast Radiosity Image Generation," *Proc. of SIGGRAPH' 88*, August 1988, pp. 75-84.

[CoGr85] Cohen, M.F. and Greenberg, D.P., "The Hemi-cube: A Radiosity Solution For Complex Environments", *Proc. of SIGGRAPH' 85*, July 1985, pp. 31-40.

[CoGr86] Cohen, M.F., Greenberg, D.P., Immel, D.S. and Brock, P.J., "An Efficient Radiosity Approach for Realistic Image Synthesis", *IEEE CG & A*, March 1986, pp. 26-35.

[CoPo84] Cook, R.L., Porter, T. and Carpenter, L., "Distributed Ray Tracing", *Computer Graphics*, Vol. 18, No. 3, July 1984, pp. 137- 145.

[Cr77] Crow, F. C., "The Aliasing Problem in Computer-Generated Shaded Images", *Comm. of ACM*, Vol. 20, No. 11, Nov. 1977, pp. 799-805.

[Crow77] Crow, F.C., "Shadow Algorithms for Computer Graphics", *Proc. of SIGGRAPH' 77*, 424-247.

[DoTo81] Doctor, L.J., Torborg, J.G., "Display Techniques for Octree-Encoded Objects", *IEEE CG & A*, July 1981, pp. 29-38.

[FoDa83] Foley, J. D. and Dam, A. V., *Fundamentals of Interactive Computer Graphics*, Addison-Wesley Publishing Company, 1983.

[FoDa90] Foley, J. D., Dam, A. V., Feiner, S.K. and Hughes, J.F., *Computer Graphics: Principles and Practice*, Second Edition, Addison-Wesley Publishing Company, 1983.

[FoFu80] Fournier, A. and Fussell, D., "Stochastic Modeling in Computer Graphics", *suppl. to Proc. SIGGRAPH 80*, August 1980, pp. 9-15.

[FoFu82] Fournier, A., Fussell, D. and Carpenter, L., "Computer Rendering of Stochastic Models", *Graphics and Image Processing*, Vol. 25, No. 6, June 1982, pp. 371-384.

[FuAb83] Fuchs, H., Abram, G.D., and Grant, E.D., "Near Real-Time Shaded Display of Rigid Objects", *Proc. of SIGGRAPH' 83*, Vol. 17, No. 3, July 1983, pp. 65-69.

[FuKe80] Fuchs, H., Kedem, Z.M., and Naylor, B.F., "On Visible Surface Generation by A Priori Tree Structures", *Proc. of SIGGRAPH' 80*, Vol. 14, No. 3, July, 1980, pp. 124-133.

[FuTa86] Fujimoto, K., Tanaka, T. and Iwata, K., "ARTS: Accelerated Ray-Tracing System", *IEEE CG&A* , April 1986, pp. 16-26.

[FuTo83] Fujimura, K., Toriya, H., Yamaguchi, K. and Kunii, T. L., " An Enhanced Oct-tree Data Structure and Operations for Solid Modeling", *Techniquical Report 83-01*, Dept. of Information Science, University of Tokyo, Jan. 1983.

[Garg82] Gargantini, I., "Linear Octree for Fast Processing of Three-Dimensional Objects", *Computer Graphics and Image Processing*, Vol. 20, No. 4, 1982, pp. 265-274.

[Gl84] Glassner, A. S., "Space Subdivision for Fast Ray Tracing", *IEEE CG & A*, 4(10), Oct. 84, pp. 15-22.

[Go71] Gouraud, H., "Computer Display of Curved Surfaces", *Ph.D. Thesis*, University of Utah, 1971.

[GoTo84] Goral, C.M., Torrance K. E., Greenberg, D. P. and Battaile, B., "Modeling the Interaction of Light Between Diffuse Surfaces", *Proc. of SIGGRAPH' 84*, July 1984, pp. 213-222.

[GrCo86] Greenberg, D. P., Cohen, M.F. and Torrance, K.E., "Radiosity: A method for Computing Global Illumination", *the Visual Computer*, Vol. 2, 1986, pp.291-297.

[Gree87] Green, M., "A Frame Buffer Package", Software manual, Department of Computing Science, University of Alberta, 1987.

[HaSa91] Hanrahan, P., Salzman, D. and Aupperle, L., "A Rapid Hierarchical Radiosity Algorithm", *Proc. of SIGGRAPH' 91*, July 1991, pp. 197-206.

[Heck91] Heckbert, P. S., *Simulating Global Illumination Using Adaptive Meshing, Ph. D. Thesis*, Computer Science Division (EECS), University of California, Berkeley, June 1991.

[HeHa84] Heckbert, P. S. and Hanrahan, P., "Beam Tracing Polygonal Objects", *Computer Graphics*, Vol. 18, No. 3, July 1984, pp. 119-127.

[ImCo86] Immel, D.S., Cohen, M.F. and Greenberg, D.P., "A Radiosity Method for Non-diffuse Environments", *Proc. of SIGGRAPH' 86*, 1986, pp. 133-142.

[JaTa80] Jackins, C. L. and Tanimoto, S. T., "Oct-trees and Their Use in Representing Three-dimensional Objects", *Computer Graphics and Image Processing*, Vol. 14, Nov. 1980, pp. 249-270.

[JoSi86] Joe, B, and Simpson, R.B., "Triangular Meshes for Regions of Complicated Shape", *International Journal for Numerical Methods in Engineering*, Vol. 23, 1986, pp. 751-778.

[Kaji86] Kajiya, J.T., "The Rendering Equation", *Proc. of SIGGRAPH' 86*, August, 1986, pp. 143-150.

[Kapl85] Kaplan, M.R., "Space-Tracing, a Constant Time Ray-Tracer", *Proc. of SIGGRAPH' 85 Tutorial*, San Francisco, July 1985.

[LaCa80] Lane, J. M.; Carpenter, L. C.; Whitted, J. T.; and Blinn, J. F., "Scan Line Methods for Displaying Parametrically Defined Surfaces", *Comm. of the ACM*, Vol. 23, No. 1, January, 1980, pp. 23-34.

[LiTa91] Lichinski, D., Tampieri, F. and Greenberg, D., "Improving Sampling and Reconstruction Techniques for Radiosity", *Tech. Report 91-1202*, Department of Computer Science, Cornell University, May 1991.

[Ma82] Mandelbrot, B. B., *The Fractal Geometry of Nature*. Freeman (1982).

[MaVa68] Mandelbrot, B. B. and Van Ness, J. W., "Fractional Brownian Motions, Fractional Noises and Applications", *SIAM Review*, 10,4, (Oct 1968), 422-437.

[NeNe72] Newell, M.E., Newell, R.G., and Sancha, T.L., "A New Approach to Shaded Picture Problem", *Proc. ACM National Conference*, 1972, pp. 443-450.

[NiNa83] Nishita, T. and Nakamae, E., "Half-Tone Representation of 3-D Objects Illuminated by Area Sources or Polyhedron Sources", *Proc. of IEEE Computer Society International Computer Software and Applications Conference (COMPSAC), IEEE Computer Society*, Washington, DC, November 1983, 237-241.

[NiNa85] Nishita, T. and Nakamae, E., "Continuous Tone Representation of Three-Dimensional Objects Taking Account of Shadows and Interreflection", *Proc. of SIGGRAPH' 85*, 23-30.

[NiOk85] Nishita, T., Okamura, E. and Nakamae, E., "Shading Models for Point and Linear Sources", *ACM TOG*, Vol. 4, No. 2, April 1985, 124-132.

[Ph73] Phone,Bui-Tuong, "Illumination for Computer Generated Images", *Ph.D. thesis*, University of Utah, 1973.

[Re83] Reeves, W. T., "Particle Systems - A Technique for Modelling a Class of Fuzzy Objects", *Computer Graphics*, Vol. 17, No. 3, July 1983, pp. 359-376.

[ReBl85] Reeves, W. T. and Blau, R., "Approximate and Probabilistic Algorithms for Shading and Rendering Structured Particle Systems", *Computer Graphics*, Vol. 19, No. 3, July 1985, pp. 313-322.

[ReGo87] Reynolds, R.A., Gordon, D., and Chen, L-S, "A Dynamic Screen Technique for Shaded Graphics Display of Slice-Represented Object", *Computer Vision, Graphics, and Image Processing*, Vol. 38, 1987, pp. 275-298.

[Ro80] Rosenfeld, A., "Quadtrees and Pyramids for Pattern Recognition and Image Processing", *Proc. of the 5th International Conference on Pattern Recognition*, Dec. 1980.

[Ro85] Rogers, D. F., *Procedural Elements for Computer Graphics*, McGraw-Hill Book Company, New York, 1985.

[SaWe88] Samet, H., Webber, R.E., "Hierarchical Data Structures and Algorithms for Computer Graphics", *IEEE CG & A*, July 1988, pp.59-75.

[Scha78] Schachter, B., "Decomposition of Polygons into Convex Sets", *IEEE Trans. on Computer*, Vol. c-27, No. 11, Nov., 1978, pp.1078-1082.

[SiBo72] Sidhu, G. S. and Boute, R. T., "Property Encoding: Applications in binary picture encoding and boundary following", *IEEE Trans. on Computes*, No. c-21, Vol. 11, Nov. 1972, pp. 1206-1216.

[SiHo78] Siegel, R. and Howell, J.R., *Thermal Radiation Heat Transfer*, Hemisphere Publishing Corp., 1978.

[SpCe78] Sparrow, E.M. and Cess, R.D., *Radiation Heat Transfer*, Hemisphere Publishing Corp., 1978.

[SpDe85] Speer, L.R., DeRose, T.D. and Barsky, B.A., "A Theoretical and Empirical Analysis of Coherent Ray Tracing", *Computer-Generated Image (Proc. of Graphics Interface' 85*), May 27-31, 1985, pp.11-25.

[ScBr69] Schumacker, R.A., Brand, B., Gilliland, M. and Sharp, W., "Study for Applying Computer-generated Images to Visual Simulation", *U.S. Air Force Human Resource Lab Tech. Rep., AFHRL-TR-69-14*, Sept. 1969, NTIS AD 700 375.

[SyBa87] Syder, J. M. and Barn, A., "Ray Tracing Complex Models Containing Surface Tessellation", *Proc. of SIGGRAPH' 87*, pp. 119.

[TaLi91] Tampieri, F. and Lichinski, D., "The Constant Radiosity Assumption Syndrome", *Eurographics Workshop on Rendering*, Spain, May 1991.

[Tamp91] Tampieri, F., *Global Illumination Algorithms for Parallel Computer Graphics, M. Sc. Thesis*, Program of Computer Graphics, Cornell University, Ithaca, January 1989.

[VeGr84] Verbeck, C.P. and Greenberg, D.P., "A Comprehensive Light-Source Description for Computer Graphics", *IEEE CG&A*, Vol. 4, No. 7, July 1984, pp. 66-75.

[WaDa90] Wang, Y. and Davis, W.A., "Octant Priority for Radiosity Image Rendering", *Proc. of Graphics Interface' 90*, May 1990, pp. 83-91.

[WaCo87] Wallace, J.R., Cohen, M.F., and Greenberg, D.P., "A Two-Pass Solution to The Rendering Equation: A Synthesis of Ray Tracing And Radiosity Methods", *Proc. of SIGGRAPH' 87*, July 1987, pp. 311-320.

[WaEl89] Wallace, J.R., Elmquist, K.A. and Haines, E.A., "A Ray Tracing Algorithm for Progressive Radiosity", *Proc. of SIGGRAPH' 89*, July 1989, pp.315-324.

[Wh78] Whitted, J. T., "A Scan Line Algorithm for Computer Display of Curved Surfaces", *Proc. of 5th Conf. on Computer Graphics and Interactive Techniques*, 1978.

[Will78] Williams, L., "Casting Curved Shadows on Curved Surfaces", *Proc. of SIGGRAPH' 78*, pp. 270-274.

[Wh80] Whitted, T., "An Improved Illumination Model for Shaded Display", *Comm. of the ACM*, Vol. 23, No. 6, June 1980, pp. 343-349.

[YaKu84] Yamaguchi, K., Kunii, T.L., Fujimura, K., "Octree-related Data Structure and Algorithms", *IEEE CG & A*, January 1984, pp. 53-59.

B45937